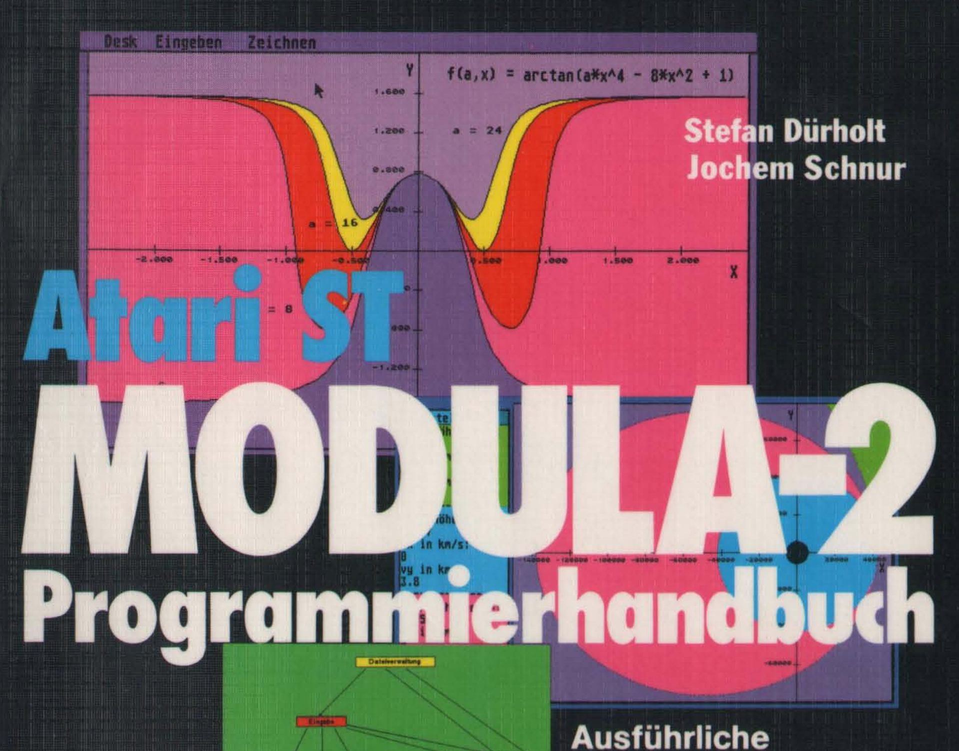
Markt&Technik



Stefan Dürholt
Jochem Schnur

Atari ST MODULA-2 Programmierhandbuch

Ausführliche Spracheinführung ★ Behandlung von Datenstrukturen ★ Nutzung von Betriebssystemroutinen ★ Einbinden von Assemblerroutinen ★ GEM-Programmierung ★ Grafik u. Sound ★ Softwareentwicklung und Modulhierarchien.

Auf zwei 3½"-Disketten enthalten:
Alle Modula-Programme in 150 Modulen teilweise kompiliert, Beispielprogramme zur Mandelbrotmenge, VDI- u. AES-Routinen, Funktionenplotprogramm u. vieles mehr



Modula-2
Programmierhandbuch



Atari ST

MODULA-2

Programmierhandbuch

Ausführliche Spracheinführung * Behandlung von Datenstrukturen * Nutzung von Betriebssystemroutinen * Einbinden von Assemblerroutinen * GEM-Programmierung * Grafik und Sound * Softwareentwicklung und Modulhierarchien.

Stefan Dürholt
Jochem Schnur

Markt&Technik Verlag AG

CIP-Titelaufnahme der Deutschen Bibliothek

Dürholt, Stefan:
Atari-ST-Modula-2-Programmierhandbuch :
ausführliche Spracheinführung ; Behandlung von Datenstrukturen ; Nutzung von Betriebssystemroutinen ;
Einbinden von Assemblerroutinen ; GEM-Programmierung ; Grafik u. Sound ;
Softwareentwicklung und Modulhierarchien / Stefan Dürholt ; Jochem Schnur. –
Haar bei München : Markt-u.-Technik-Verl., 1989
ISBN 3-89090-775-X
NE: Schnur, Jochem:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

93 92 91 90

ISBN 3-89090-775-X

Einbandgestaltung: Grafikdesign Heinz Rauner
Dieses Produkt wurde mit Desktop-Publishing-Programmen erstellt
und auf der Linotronic 300 belichtet
Druck: Schoder, Gersthofen
Printed in Germany

Inhaltsverzeichnis

Anhang 483

VORWORT

Die Programmiersprache Modula-2 wurde Ende der 70er Jahre als direkte Nachfolgerin der Sprachen Pascal (1970) und Modula (1975) an der Eidgenössischen Technischen Hochschule Zürich unter der Leitung von Professor *Niklaus Wirth* entworfen. Die Zielsetzung eines im Jahre 1977 begonnenen Forschungsprojektes war es, ein Rechnersystem (Hard- und Software) in einem einheitlichen Ansatz zu entwickeln. Die Sprache sollte daher sowohl der Systementwicklung auf hoher Ebene, als auch den Anforderungen auf niedriger, maschinennaher Ebene gerecht werden. Die erste Implementierung wurde 1979 fertiggestellt, 1980 erfolgte die Veröffentlichung der Sprache Modula-2.

Modula-2 ist Pascal recht ähnlich, was dem Pascal-Programmierer den Umstieg erleichtern dürfte. Die Erweiterungen und Verbesserungen gegenüber Pascal sind aber gravierend. Sie lassen sich in vier Punkten zusammenfassen:

- Modulkonzept
- maschinennahe bzw. systemnahe Elemente
- Prozedurtyp
- syntaktische Straffungen

Pascal hatte einige Nachteile bei der systemnahen Programmierung, zum Beispiel waren keine direkten Speicherzugriffe möglich. Bei der Entwicklung großer Programme bereitete es Schwierigkeiten, daß man diese nicht in kleinere, übersichtliche Einheiten zum getrennten Übersetzen und Austesten zerlegen konnte. Außerdem fehlte die Möglichkeit, bereits vorhandene Funktionen aus »Bibliotheken« einzubinden (beides war in FORTRAN schon üblich, ähnlich wie in C). Da dies aber oft unumgänglich ist, bieten verschiedene Hersteller eigene Pascal-Dialekte an, die mit dem Standard-Pascal oft wenig gemeinsam haben.

Pascal konnte sich auf dem Atari-ST nicht so recht durchsetzen. Dahingegen gibt es seit kurzem mehrere Modula-2-Compiler, die zum Teil direkt von dem von *Wirth* konzipierten adaptiert worden sind. Mit Modula-2 ist dem Atari-ST-Benutzer eine Sprache mit sehr hohem Niveau gegeben, mit der er dennoch systemnah programmieren kann. Es ist daher nicht verwunderlich, daß das Interesse an Modula-2 sprunghaft angestiegen ist. Die Vormachtstellung von Pascal im Lehrbereich sowie die von »C« im Bereich der Systemprogrammierung wird angegriffen, was nur verständlich ist, da Modula-2 die Stärken beider Sprachen vereint.

Modula-2 wurde speziell zur Entwicklung großer Programmprojekte entworfen. Es unterstützt die Möglichkeit der schrittweisen, strukturierten Programmerstellung in separaten Einheiten, den sogenannten »Modulen«, die getrennt übersetzt, geprüft und anschließend zu einem Programm zusammengefügt (»gelinkt«) werden. Die Aufrufe der System-Routinen

sind einfach über die mitgelieferten Module möglich, wodurch eine Programmierung unter GEM leicht realisiert wird. Der Benutzer hat mit den zum Teil integrierten Debuggern und Assemblern brauchbare Entwicklungspakete zur Verfügung, die keinen Vergleich zu scheuen brauchen.

Das vorliegende Buch über Modula-2 berücksichtigt sowohl den Leser, der sich in diese Sprache einarbeiten will und nur wenige oder keine Programmierkenntnisse besitzt, geht aber in starkem Maße auch auf den erfahrenen Programmierer ein, der mit den Sprachen Basic, C, Pascal oder Modula-2 vertraut ist.

Ziel ist es, jeweils mit grundlegenden Programmiertechniken beginnend, den Leser schnell an professionelle Module für den jeweiligen Themenkreis heranzuführen, so daß in jedem Fall fortgeschrittene, optimierte Routinen geboten werden. Das Konzept ist also, den Leser schnell von »Null« auf »High-Level«-Programmierung zu bringen.

Der Stil der Beschreibung ist dabei knapp gehalten, aber ausreichend informativ und enthält konkrete Hinweise auf die Programmierfeinheiten (z.B. Geschwindigkeitsoptimierung). Insgesamt erhält der Leser neben einer nach didaktischen Prinzipien erstellten Einführung eine Fülle von allgemein nützlichen Routinen und Modulen, die auch der professionelle Modula-2-Programmierer für die Einbindung in eigene Programme zu schätzen wissen wird. Die Programmentwicklung wird durch die Prozedurenbibliothek, die dieses Buch bietet, bequemer und zeitsparender.

Die Grobgliederung des Buches ist folgende:

Kapitel 1: Spracheinführung

Kapitel 2: Behandlung wichtiger Datenstrukturen

Kapitel 3: Benutzung des 68000-Assemblers unter Modula-2

Kapitel 4: GEM-Programmierung unter Modula-2

Kapitel 5: Demonstration der Entwicklung eines komplexen Programmprojekts

Die ersten beiden und das letzte Kapitel sind für jeden Modula-2-Programmierer von Interesse, unabhängig davon, mit welchem Rechner er arbeitet. Kapitel 3 spezialisiert sich auf Computer mit dem 68000-Prozessor. Das 4. Kapitel enthält hauptsächlich Atari-spezifische Eigenheiten. Im einzelnen:

Zu Kapitel 1

Nach einführenden Beispielen werden die vielfältigen Datenstrukturen von Modula-2 besprochen. Hierbei ist es didaktisches Prinzip, zu erläutern, wie die Daten im Speicher abgelegt

werden. Dies erleichtert das Verständnis der Datenstrukturen und ist besonders bei der Behandlung von Zeigern vorteilhaft.

Der eilige und erfahrene Pascal-Programmierer wird diese beiden Abschnitte und die folgenden drei über Standard-Prozeduren, Kontrollstrukturen sowie das Prozeduren-Konzept sicherlich rasch lesen können. Auch kann er sich zunächst einen Überblick über die Unterschiede von Pascal und Modula-2 verschaffen. Neu dürfte für ihn jedoch das Modul-Konzept und die Bearbeitung paralleler Prozesse sein. Für den Anfänger ist das intensive Durcharbeiten des gesamten Kapitels 1 ein Muß!

Zu Kapitel 2

Hier werden komplexere Datenstrukturen wie Felder, Verbunde, dynamische Datenstrukturen und Dateien behandelt.

Die einzelnen Abschnitte sind dabei so aufgebaut, daß sie dem Neuling in leicht verständlicher Form eine Einführung geben, die dann aber rasch mit Programmierfeinheiten gewürzt wird und dann ohne Umschweife zu Prozeduren führt, die auch dem Profi nützliches Handwerkszeug für seine Programme bieten.

Bei der Behandlung der verzeigerten Strukturen wird großer Wert darauf gelegt, dem Leser die Darstellung der Daten auf dem »Heap« im Speicher vor Augen zu führen. Die üblichen Programmierfehler, die bei Zeigern oft auftreten, dürften sich so weitgehend vermeiden lassen.

Zu Kapitel 3

Dieses kurze Kapitel zeigt die Benutzung des 68000-Assemblers von Modula-2 aus, die sich dann anbietet, wenn es um die Optimierung geschwindigkeitskritischer Routinen geht.

Für den in 68000-Assembler unerfahrenen Leser wird im ersten Abschnitt die Einführung in die Befehlsklassen des Prozessors gegeben. Dieser Teil wird durch eine Tabelle im Anhang zum Nachschlagen abgerundet. Die Assemblerkenntnisse werden zur Erstellung eigener Module genützt (Spracherweiterung).

Zu Kapitel 4

Im 1. Abschnitt geht es darum, den Atari-Neuling mit den Betriebssystem-Routinen und dem GEM vertraut zu machen. Dem auf dem Atari erfahrenen C-, Pascal- oder Assembler-Programmierer nützen hier die Ausführungen über die verschiedenen Module. Es wird also das Dickicht der zahlreichen Routinen geordnet nach dem Prinzip »Was benötige ich für welchen Zweck?«.

Im folgenden werden Anwendungen zur Fenstertechnik, Menütechnik und Grafik gezeigt. Der Abschnitt über Grafik wird neben einem vielseitig nutzbaren Modul mit Grafikprozedu-

ren durch einige interessante Programme über Mandelbrot-, Juliamengen sowie Simulationsprogrammen – unter anderem zu Satellitenbahnen – abgerundet.

Es geht hier also nicht um die in der Literatur zum Atari weitverbreitete Aufzählung der Systemroutinen, sondern es soll in exemplarischen Anwendungssituationen gezeigt werden, wie man sie einsetzt.

Zu Kapitel 5

Das Abschlußkapitel zeigt ein größeres Programmprojekt auf. Es geht um ein ausgefeiltes Funktionenplotprogramm. Auch hierbei werden in einzelnen Teilabschnitten theoretisch anspruchsvolle Sachverhalte schrittweise klargemacht.

Im einzelnen geht es um das »Scannen« und »Parsen« von Termen als Zeichenketten (grenzt etwa an Compilerbau), die Bildung der Ableitung einer Funktion als Zeichenkette, Optimierung der Integration sowie grafische Darstellung von Funktionen. Hier werden Methoden der KI-Programmierung (künstliche Intelligenz) genutzt. Das fertige Programm zeigt beispielhaft, wie man unter GEM eine sinnvolle Benutzerführung programmiert (Menütechnik, Maskeneingabe, Ausgabe in Fenstern u. ä.). Außerdem geht es um die Anwendung der in den vorangegangenen Kapiteln vorgestellten Hilfsmodule, so daß insgesamt interessantes Anschauungsmaterial für die Behandlung eines größeren Programmpaketes unter Modula-2 demonstriert wird.

Die Quelltexte sämtlicher Programme dieses Buches befinden sich auf den beiliegenden Disketten. Die 150 Files sind in 5 Ordnern kapitelweise gegliedert. Auf beiden Disketten finden Sie »LIES-MICH-Dateien«, die über den Inhalt und die Handhabung Aufschluß geben.

Abschließend möchten wir Herrn Hans Helmut Hager und Herrn Alfred Rodenbücher für die Durchsicht unseres Manuskripts, sowie Carmen für die Geduld, die sie uns während der Arbeit an diesem Buch entgegengebracht hat, danken.

Wir wünschen dem Leser viel Freude bei der Lektüre dieses Buches sowie bei der Erstellung eigener Modula-Programme!

Jochem Schnur
Stefan Dürholt

KAPITEL 1

Einführung in die Programmiersprache Modula-2

1.1 Die ersten Schritte

1.1.1 Installierung des Modula-Entwicklungspakets

Das gesamte erste Kapitel sollten Sie gründlich durcharbeiten, bevor Sie ihr erstes Modula-2-Programm schreiben. Da Sie diesen Rat sowieso nicht befolgen, beginnen wir gleich mit drei einfachen Programmbeispielen.

Zuvor müssen Sie jedoch das Modula-Entwicklungspaket installieren.

Wir haben bei der Arbeit an diesem Buch folgende Systeme verwendet:

1. *Hänisch-Modula-2* der Firma *Rolf Hänisch Software,* Berlin
2. *Megamax Modula-2* der Firma *Application Systems Heidelberg*
3. *MSM2* der Firma *Modular Software, Firnau und Krey,* Kronshagen
4. *SPC-Modula-2* der Firma *advanced applications Viczena,* Karlsruhe
5. *TDI-Modula-2/ST* der Firma *Modula-2 Software Ltd.,* GB-Bristol

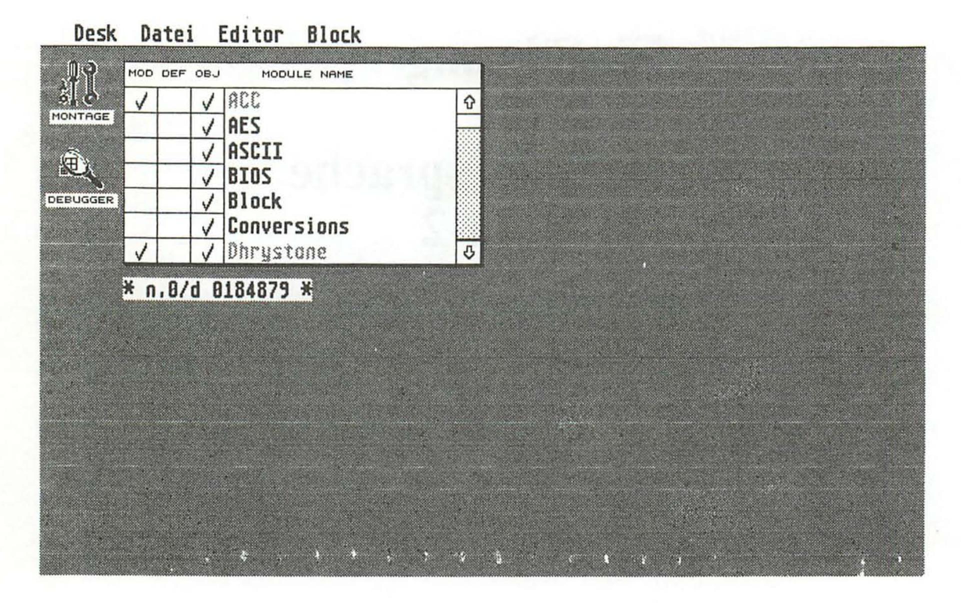


Bild 1.1: Hänisch-Modula-2

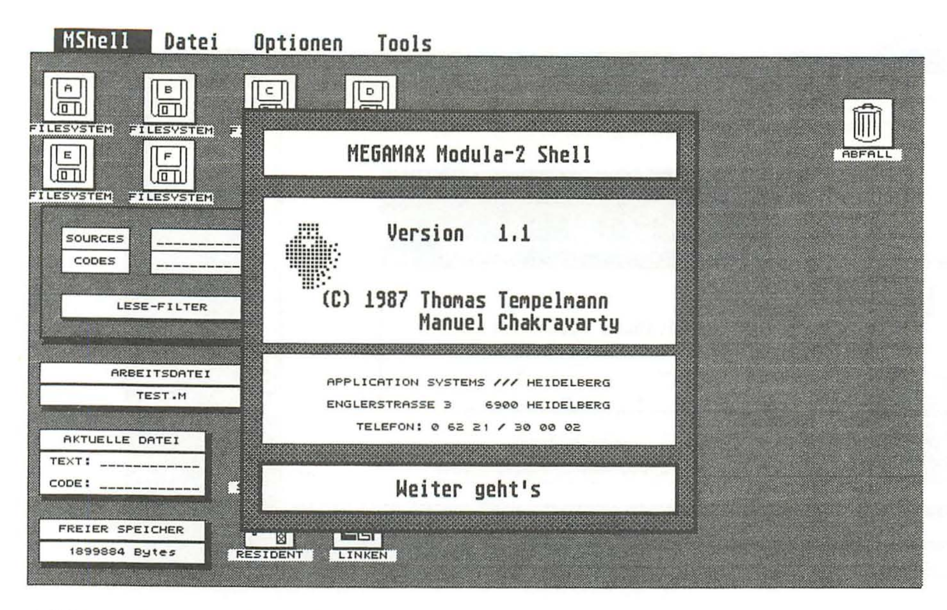


Bild 1.2: Megamax Modula-2

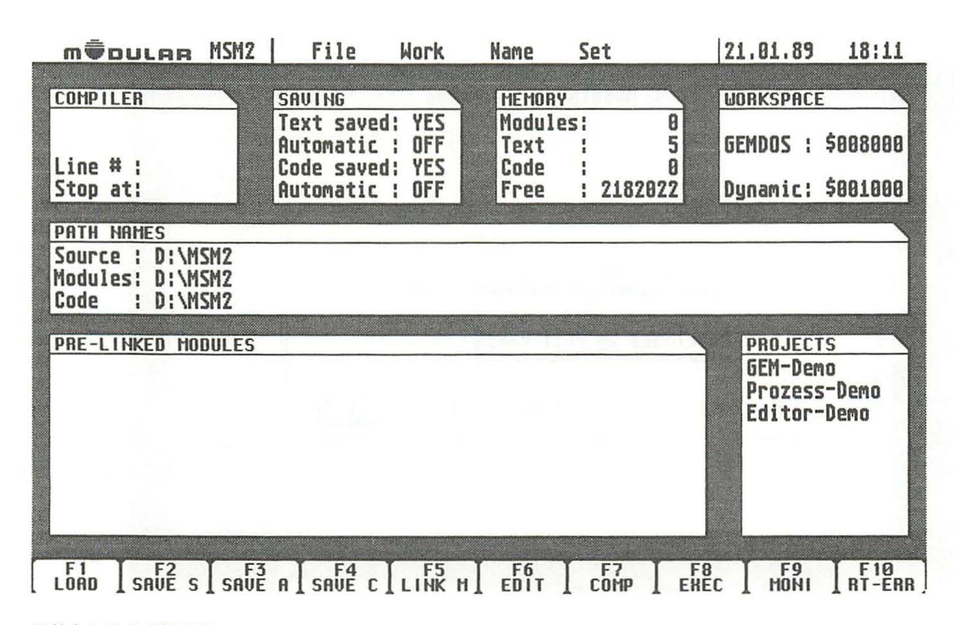


Bild 1.3: MSM2

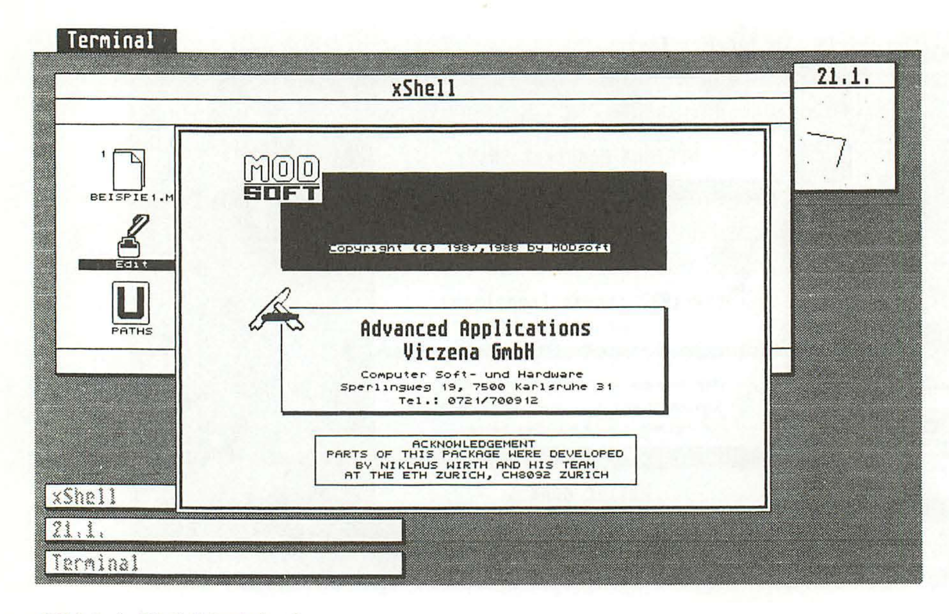


Bild 1.4: SPC-Modula-2

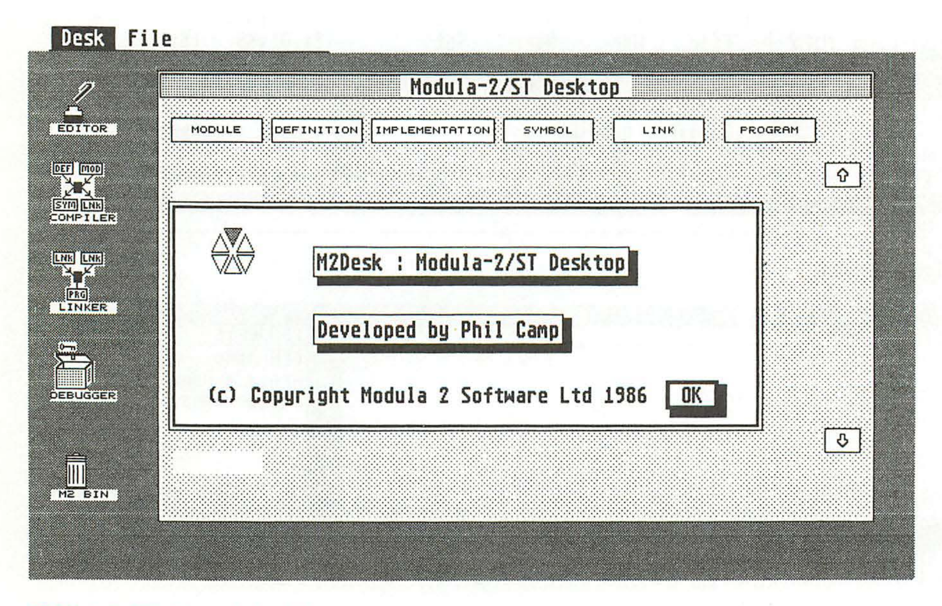


Bild 1.5: TDI-Modula-2/ST

Alle Modula-2-Systeme für den Atari arbeiten mit einem komfortablen Rahmenprogramm, einer sogenannten »Shell«. Dieses Programm wird vom »Desktop« (so nennt sich die Arbeitsfläche, die Sie auf dem Bildschirm sehen, wenn Sie den Atari einschalten) aus gestartet, und Sie können nun von hier aus den Editor, Compiler und Linker mit der Maus anwählen.

Der **Editor** dient dazu, Programmtexte einzutippen und abzuspeichern. Verschiedene Funktionen des Editors helfen bei der Arbeit, beispielsweise beim Verschieben von Textblöcken oder beim Suchen von Wörtern. Die meisten Funktionen lassen sich über die Menüleiste mit der Maus anwählen. Schauen Sie sich die verschiedenen Punkte an; ihre Kenntnis ist für ein effektives Arbeiten wichtig.

Der **Compiler** hat die Aufgabe, aus dem Programmtext nun Code zu erzeugen, der auf dem Atari ablauffähig ist. Nach fehlerfreier Übersetzung – das heißt, wenn der Programmtext in Ordnung war – können Sie Ihr Programm sofort von der Shell aus laufen lassen und austesten.

Direkt vom Desktop (ohne die Shell) ist das Programm so nicht zu starten, da es Funktionen benötigt, die ihm vom »Laufzeit-System«, wozu auch die Shell gehört, geliefert werden müssen. Werden dem Programm aber die Funktionen direkt eingebaut, so kann es auch ohne die Shell auskommen. Dieses Zusammenbauen übernimmt der **Linker** (= Binder).

Neben dem Dreiergespann Editor – Compiler – Linker gibt es noch andere nützliche Einrichtungen in der Shell, die Sie bei der Entwicklungsarbeit unterstützen können. So haben Sie innerhalb der Shell die Möglichkeit, mit Dateien umzugehen, ähnlich wie Sie es vom Desktop her gewohnt sind. Man kann sich innerhalb der Shell das Inhaltsverzeichnis von Disketten ansehen und Dateien löschen oder kopieren. Normalerweise werden Sie aber den Editor oder Compiler auf die Dateien loslassen, so daß Sie während der Entwicklung von Programmen die Shell kaum verlassen müssen.

Um optimal mit der Modula-Shell arbeiten zu können, ist eine einmalige Voreinstellung des gesamten Systems nötig. Diese Voreinstellung ist abhängig von der Hardwarekonfiguration Ihres Arbeitsplatzes. Hierbei spielt die Größe des Speichers und das Vorhandensein einer Festplatte eine Rolle.

Da Diskettenzugriffe auf dem Atari relativ lange dauern, ist es sinnvoll, bestimmte Teile des Systems permanent im Speicher zu halten. Dies kann auf einer RAM-Disk geschehen. Damit zu Beginn einer Arbeitssitzung stets die gleichen Teile permanent geladen sind, gibt es die Möglichkeit, diese Voreinstellungen in einer Informationsdatei festzuhalten. Bei einem neuerlichen Start der Shell wird diese Informationsdatei gelesen und die Arbeitsumgebung entsprechend konfiguriert. In dieser Datei werden auch die Suchpfade für Ihre edierten Texte und die übersetzten Module festgehalten.

Bitte haben Sie Verständnis dafür, daß wir hier nicht die Installationsprozedur im einzelnen vorstellen können. Sie ist, wie gesagt, von Ihrer Hardware und vom jeweiligen Modula-System

abhängig. Das Handbuch zu Ihrem System gibt Hinweise für die Voreinstellungen, mit denen Sie effektiv arbeiten können. Eine sehr benutzerfreundliche Methode zur Installation ist bei SPC-Modula gegeben: Hier kann man ein Programm namens `INSTALL.PRG` starten, das gesteuert nach Benutzereingaben, die Installation vornimmt.

1.1.2 Erste Beispiele

Sie haben es sicher geschafft, von der Shell aus den Editor zu starten. Tippen Sie nun folgenden Text und achten Sie beim Abschreiben unbedingt auf Groß- und Kleinschreibung. Wenn Sie allerdings keine Lust haben, den Text selber einzugeben: Er befindet sich auch auf der Diskette und hat den File-Namen `BEISPIE1.M` und kann in den Editor geladen werden.

```
MODULE ErstesBeispiel;

FROM InOut IMPORT WriteString, WriteLn;

VAR i : CARDINAL;                                      (* zum Zählen *)

BEGIN
  FOR i:=1 TO 100 DO                             (* läuft für i:=1,2..100 *)
    WriteString("Hurra, mein erstes Modula-Programm läuft!");
    WriteLn                                            (* Zeilenvorschub *)
  END
END ErstesBeispiel.
```

Verlassen Sie den Editor und kompilieren Sie den Text. Wenn Sie nicht das Megamax-System benutzen, sollten Sie die Datei zuvor in `BEISPIEL.MOD` umbenennen. Das gilt für alle Module dieses Buchs und kann vom Desktop aus mit dem Menüpunkt »`zeigeInfo...`« erledigt werden.

Falls der Compiler Fehler findet, hat man die Möglichkeit, in den Editor zurückzukehren. Der Cursor steht auf der Fehlerstelle. Im Anschluß an das erfolgreiche Ausmerzen der Fehler starten Sie das Programm.

Nach diesem ersten Erfolgserlebnis sollten wir uns den Programmtext einmal genauer anschauen. Das erste Wort heißt `MODULE`, das wie alle »Schlüsselwörter« der Sprache groß geschrieben wird (siehe 1.2.2). `MODULE` kennzeichnet den Programmanfang (für Pascal-Programmierer: es ersetzt das Wort `PROGRAM`). Dahinter folgt ein »Bezeichner«, das ist ein Name, den Sie sich in gewissen Grenzen aussuchen dürfen. In einem Bezeichner darf kein Leerzeichen stehen. Man macht deshalb von Groß- und Kleinschreibung Gebrauch, um ihn aus meh-

reren Wörtern zusammenzusetzen. Die weiteren Spielregeln werden im Abschnitt 1.2.1 erläutert.

Anschließend folgt eine »Importliste«, in der angegeben wird, von welchen schon existierenden Modulen Teile »importiert« werden sollen. Die Sprache Modula-2 – im folgenden auch kurz »Modula« genannt – selbst hat selbst keine Funktionen zur Eingabe und Ausgabe. Das wird denjenigen verwundern, der Programmiererfahrung in Basic oder Pascal hat. Überwinden Sie diesen Kulturschock! Es hat durchaus Vorteile, daß die Ein- und Ausgaberoutinen nicht mehr von der Programmiersprache selbst festgelegt sind, sondern aus mitgelieferten externen Modulen abgerufen werden können. Module können nämlich ausgetauscht werden, auch durch selbstgeschriebene. Diesen Vorteil werden Sie später noch zu schätzen wissen. Ein weiteres Plus ist die getrennte Übersetzbarkeit. Module können separat entwickelt, ausgetestet und kompiliert werden. Man braucht nicht immer alles mit zu übersetzten, was schon längst läuft. Der Code kompilierter Module wird dem neuen Programm einfach »hinzugelinkt«. Module sind also flexible Programmbausteine. Haben Sie also keine Angst, gleich zu Beginn mit ihnen Bekanntschaft zu machen. Ohne Module läuft in Modula nämlich nichts!

Kommen wir nach diesen generellen Bemerkungen auf unser Programm zurück. Ein- und Ausgabeprozeduren kann man sich einfach holen: es gibt sie unter anderem im Standardmodul `InOut`, der zu jedem Modula-System mitgeliefert wird. Wir benötigen `WriteString` um eine Zeichenkette (= »String«) auf dem Bildschirm zu schreiben und `WriteLn` (»Write Line«) für einen Zeilenvorschub, also um eine neue Zeile zu beginnen.

Als nächstes brauchen wir eine Variable zum Zählen. Ihr Name ist `i` und ihr »Datentyp« ist `CARDINAL`, was die natürlichen Zahlen 0, 1, ... 65535 beinhaltet. Es folgt der eigentliche Programmtext, welcher zwischen `BEGIN` und `END`<Modulname> gefolgt von einem Punkt steht. Unser Programm besteht aus einer Wiederholungsanweisung, einer »`FOR`-Schleife«. Zunächst wird die Variable `i` auf `1` gesetzt und der Schleifenrumpf (das, was hier zwischen dem `FOR` und dem dazugehörenden `END` steht) ausgeführt. Der Rumpf besteht hier aus einer Textausgabe gefolgt von einem Zeilenvorschub. Anschließend wird `i` um 1 erhöht – das macht die `FOR`-Schleife automatisch – und das ganze läuft von vorne ab. Zum letztenmal, wenn `i` gleich 100 ist. Das erste `END` markiert das Ende der Schleife, dann folgt die Programmende-Markierung. Wenn Sie Pascal kennen, fällt auf, daß der Schleifenrumpf nicht mit `BEGIN` und `END` geklammert ist. In Modula braucht man hier kein `BEGIN`, da `FOR` in Modula immer ein `END` verlangt.

Text, der zwischen »`(*`« und »`*)`« steht, wird vom Compiler nicht beachtet. Man kann dort einen beliebigen Kommentar einfügen.

Auch im nächstem Programm geht es um die Ausgabe von Zeichenketten, aber außerdem noch um Eingaben:

```
MODULE ZweitesBeispiel;

FROM InOut IMPORT WriteString, WriteLn, ReadString, Read;

VAR antwort : CHAR;                                   (* ein Zeichen *)
    name    : ARRAY [0..79] OF CHAR;      (* Zeichenkette mit 80 Zeichen *)

BEGIN
  WriteString("Geben Sie bitte Ihren Namen ein : ");
  ReadString(name);
  REPEAT                                  (* Wiederhole das Folgende... *)
    WriteLn; WriteLn;
    WriteString("Hallo, "); WriteString(name); WriteString("!");
    WriteLn;
    WriteString("Wir wünschen Ihnen viel Freude");
    WriteLn;
    WriteString("beim Durcharbeiten dieses Buches.");
    WriteLn; WriteLn;
    WriteString("Noch einmal (j/n) : ");
    Read(antwort);
  UNTIL antwort = "n"                     (* ... bis 'antwort' = "n" ist *)
END ZweitesBeispiel.
```

Hier wird der Benutzername (Variable `name`) und später (gegen Ende) ein einzelnes Zeichen (`antwort`) eingegeben. Um ein einzelnes Zeichen zu speichern, braucht man eine Variable vom Datentyp `CHAR` (engl. *character* = dt. »Zeichen«). Die Zeichenkette `name` besteht aus mehreren Zeichen (im Beispiel maximal 80). Man spricht von einem »Feld« von 80 Zeichen. Die Zeichen sind durchnumeriert, hier von 0 bis 79 (macht 80). Also definiert man in Modula

```
VAR name: ARRAY [0..79] OF CHAR.
```

Die Zeichenkettenvariable `name` wird mit der Prozedur `ReadString` eingegeben. An dieser Stelle soll der Benutzer seinen Namen eintippen, der dann als Inhalt in der Variable `name` steht. Für das Einlesen eines einzelnen Zeichens nimmt man die Prozedur `Read`. Es läuft wieder eine Schleife ab, diesmal aber solange, bis der Benutzer ein »n« für nein eintippt. Es handelt sich um eine »wiederhole <*Block*> bis <*Bedingung*>-Schleife« oder in Modula `REPEAT`<*Block*> `UNTIL`<*Bedingung*>. Der <*Block*> wird wiederholt, bis die Bedingung erfüllt ist. An dem kleinen Programm erkennt man, daß zwischen zwei Anweisungen ein Semikolon als Trennzeichen gesetzt wird. Vor `END` und `UNTIL` darf es entfallen.

Im nächsten (und letzten) Einführungsbeispiel wollen wir den »größten gemeinsamen Teiler« (`ggT`) zweier einzugebenden natürlichen Zahlen `i` und `j` berechnen lassen:

```
MODULE DrittesBeispiel;

FROM InOut IMPORT ReadCard, WriteCard, WriteString, WriteLn, Read;

VAR i,j,ggT : CARDINAL;
    antwort : CHAR;

BEGIN
  WriteString("Programm zur Berechnung des größten ");
  WriteString("gemeinsamen Teilers (ggT) zweier Zahlen");
  REPEAT
    WriteLn; WriteLn;
    WriteString("Geben Sie die erste  Zahl ein: "); ReadCard(i);
    WriteString("Geben Sie die zweite Zahl ein: "); ReadCard(j);
    WHILE i # j DO
      IF i > j THEN i := i - j ELSE j := j - i END      (* ggT errechnen *)
    END;
    ggT := i;
    WriteString("Der ggT dieser Zahlen lautet:  ");
    WriteCard(ggT,6);
    WriteLn; WriteLn;
    WriteString("Wünschen Sie noch eine Berechnung (j/n)? ");
    Read(antwort);
    antwort := CAP(antwort);                  (* Umwandlung in Großbuchstaben *)
  UNTIL antwort = "N";
END DrittesBeispiel.
```

Das Programm ist schon etwas luxuriöser. Es schreibt zunächst eine Überschrift. Dann kann man die beiden Zahlen i und j eingeben (mit `ReadCard`, etwa »Lese Zahl vom Typ `CARDINAL`«). Danach wird der `ggT` berechnet. Das Ergebnis wird mit `WriteCard` ausgegeben.

`WriteCard(ggT, 6)` bedeutet, daß `ggT` in einem Feld der Länge 6 rechtsbündig ausgegeben wird. Eventuell werden Leerzeichen vorangestellt. Hat die Zahl mehr als 6 Stellen, wird sie trotzdem vollständig ausgegeben. In Megamax-Modula erzeugen alle Eingabe-Prozeduren wie `ReadCard` (außer der Prozedur `Read`) einen Zeilenvorschub. Bei anderen Modula-Systemen ist das nicht der Fall. Sie müssen noch `WriteLn` im Programm dahintersetzen, damit der Bildschirm einigermaßen vernünftig aussieht!

Damit unsere Schleife auch abbricht, wenn der Benutzer sowohl ein kleines »n« als auch ein großes »N« eingibt, wandeln wir das gelesene Zeichen `antwort` mittels der Standardfunktion `CAP` in einen Großbuchstaben um; wir brauchen dann nur noch auf »N« zu testen. Bleibt noch die eigentliche Berechnung des `ggT` zu erklären:

Wenn die beiden Zahlen i und j gleich sind, ist der ggT schon gefunden, nämlich i (oder j, sie sind ja gleich). Wenn i und j ungleich sind (in Modula: `i#j` oder `i<>j`), zieht man die kleinere Zahl von der größeren Zahl solange ab, bis diese Gleichheit erreicht ist. Also:

```
Falls i > j, dann neues i durch i - j ersetzen,
sonst neues j durch j - i ersetzen.
```

Für die Umsetzung in Modula werden die englischen Wörter »`IF`« (für »falls«) und »`ELSE`« (für »sonst«) benutzt.

Die Ersetzung von `i` durch `i - j` nennt man eine »Zuweisung«. Das Zeichen dafür ist »`:=`«, lies »wird zu«. Die Schleife, die diese `IF`-Anweisung umfaßt, soll zu Beginn die Bedingung `i#j` prüfen, hierzu dient die Wiederholungsanweisung:

```
WHILE <Bedingung> DO <Anweisungen> END
```

Etwa: solange Bedingung erfüllt, führe Anweisungen aus.

Ein solches Rezept wie dieses Beispiel zur Ermittlung des `ggT` nennt man einen »Algorithmus« (etwa: »Bearbeitungsvorschrift«). Das Erstellen von Algorithmen und die Prüfung auf ihre Korrektheit ist eine wichtige Aufgabe in der Informatik. Der `ggT`-Algorithmus funktioniert dann nicht, wenn entweder für `i` oder `j` eine `0` (Null) eingegeben wurde, da dann in der Schleife die Zahlen nicht kleiner werden und so die Gleichheit nicht erreicht wird. Die Schleife würde bei einer solchen Eingabe nicht zum Ende kommen. Da bleibt dann nur noch der Griff zum Reset-Knopf! Solche Fehleingaben müssen in einem professionellen Programm durch bessere Eingabeprozeduren verhindert werden.

Nach diesen Beispielen tauchen bestimmt einige Fragen auf, etwa:

- Wie sieht allgemein die Struktur eines Moduls aus?
- Wie dürfen Bezeichner aussehen (aus welchen Zeichen dürfen sie bestehen)?
- Welche »elementaren« Datentypen gibt es in Modula (zum Beispiel `CHAR`, `CARDINAL`) und wie kann man daraus weitere Datentypen gewinnen (»strukturierte Datentypen«, zum Beispiel `ARRAY [0..79] OF CHAR`)?
- Welche reservierten Wörter gibt es in Modula (zum Beispiel `MODULE`, `IMPORT`, `END`)?
- Über welche Standardprozeduren verfügt Modula (zum Beispiel `CAP`)?
- Welche Wiederholungsanweisungen (Schleifen, zum Beispiel `REPEAT... UNTIL`) und welche bedingten Anweisungen (zum Beispiel `IF... THEN... ELSE`) gibt es? Wiederholungsanweisungen und bedingte Anweisungen kontrollieren den Programm-Ablauf. Sie werden daher unter dem Oberbegriff »Kontrollstrukturen« zusammengefaßt.

- Mit welchen Operatoren (zum Beispiel `+`, `:=`, `#`, `=`) arbeitet man in Modula?
- Welche Bibliotheksmodule (zum Beispiel `InOut`) liefert ein Modula-System?

Antworten darauf geben die nächsten Abschnitte. Darüber hinaus folgen weitere Beispiele mit neuen Algorithmen.

Nun kann man eine Spracheinführung sicherlich so gestalten, daß man den Leser von Programmbeispiel zu Programmbeispiel führt und immer weitere Besonderheiten erklärt. Das kann sicherlich recht kurzweilig sein (oder auf die Dauer langweilig), aber der Leser erhält keinen Überblick. Die Sprachbeschreibung gerät umfangreich. Alternativ hierzu könnte man zunächst einen Überblick geben, damit der Anfänger schon einmal mit dem groben Rahmen vertraut gemacht wird. Dabei wird vielleicht nicht sofort alles klar; es müssen Anwendungsbeispiele folgen. Diese Methode erfordert jedoch anfangs eine höhere »Frustrationstoleranz« beim Leser. Dafür ist sie aber schneller, kompakter und man hat gleich in der Übersicht etwas zum Nachschlagen.

Aus dieser nicht unbedingt objektiven Beschreibung beider Methoden dürfte klar geworden sein, daß wir im Unterschied zu den meisten Computer-Büchern den zweiten Weg einschlagen. Es folgt also jetzt eine »geballte Ladung« von Listen (Schlüsselwörter, Standardtypen, Operatoren usw.). Sie können sich anschließend bei den zahlreichen Programmbeispielen wieder erholen. Statt wie in anderen Einführungsbüchern die Sprachbeschreibung auf etwa 300 Seiten auszudehnen, wollen wir mit deutlich weniger auskommen und noch etliches für die professionelle Programmierung an den Mann/die Frau bringen.

1.1.3 Syntaxdiagramme

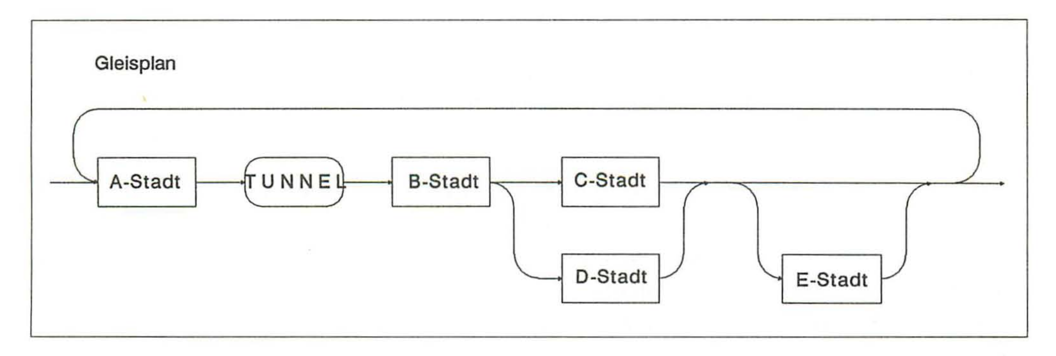


Bild 1.6: Syntaxdiagramm Gleisplan

Betrachten wir folgenden Gleisplan:

Ein von links kommender Zug passiert auf jeden Fall A-Stadt, den Tunnel und B-Stadt (Sequenz oder Zusammensetzung), hat dann die Möglichkeit, entweder C-Stadt oder D-Stadt zu passieren (Selektion oder Auswahl), kann dann E-Stadt durchqueren oder an ihr vorbeifahren (Option) und dann an der letzten Weiche das ganze beliebig oft (oder gar nicht) wiederholen (Iteration oder Wiederholung).

Eine mögliche Zugfahrt wäre also →A-Stadt →TUNNEL →B-Stadt →D-Stadt →A-Stadt →TUNNEL →B-Stadt →C-Stadt →E-Stadt →

Das Kästchensymbol A-Stadt könnte wieder so aussehen:

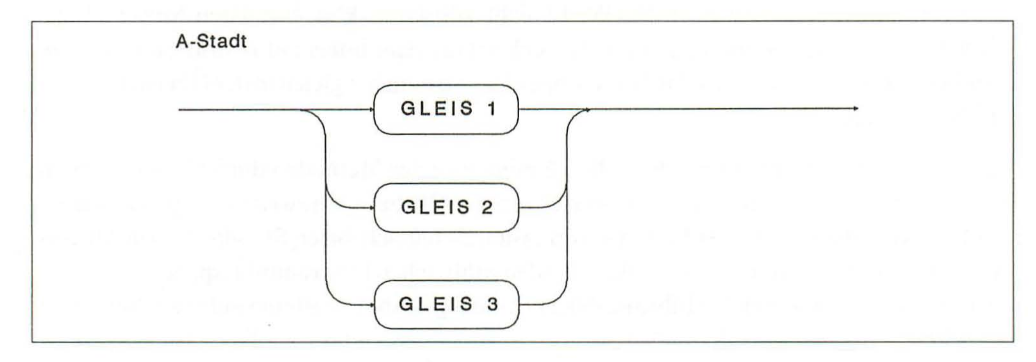
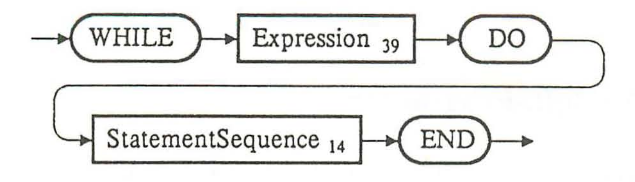


Bild 1.7: A-Stadt

Sie werden bemerkt haben, daß wir eckige und abgerundete Symbole benutzt haben. Die abgerundeten Symbole sind nicht weiter zu gliedernde Einheiten. Man spricht von Terminalsymbolen. Ein eckiges Symbol muß aber noch weiter gesondert beschrieben werden. (Non-terminal-Symbol). Es stellt also einen Verweis auf einen weiteren Gleisplan dar. Diese Beschreibung kann in mehreren Stufen geschehen, bis man schließlich auf Terminalsymbole trifft.

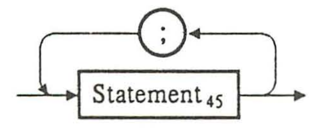
Eigentlich wollten Sie ja nicht Eisenbahner werden, sondern Modula lernen. Aber genau mit diesen »Eisenbahndiagrammen« läßt sich die Sprache Modula beschreiben! Man spricht hier nur vornehmer von »Syntaxdiagrammen«, da sie den Satzbau der Sprache festlegen.

Die Terminalsymbole sind dabei einzelne Zeichen wie Klammer, Satzzeichen, Buchstaben, Ziffern und die Schlüsselwörter der Sprache (zum Beispiel `MODULE`, `BEGIN`, `END`). Sie können nun die Syntaxdiagramme lesen, denn außer den oben geschilderten Fällen Sequenz, Selektion, Option und Iteration kann nichts weiteres vorkommen. Beachten Sie aber, daß die »Züge« nur einen Vorwärtsgang haben. Die Form der »Weichen« ist also wichtig; die Pfeile verschaffen zusätzliche Klarheit. Als Beispiel das Syntaxdiagramm der `WHILE`-Schleife:



SYNTAX: "WhileStatement"(49)

Man sieht die Schlüsselwörter `WHILE`, `DO` und `END`. »Expression« (Ausdruck) und »StatementSequence« (Anweisungsfolge) sind noch näher zu beschreiben. Dies ist für die Anweisungsfolge einfach:



SYNTAX: "StatementSequence"(14)

Die Detailbeschreibung ist noch nicht zu Ende. Wir brechen hier aber ab, da die Benutzung der `WHILE`-Schleife ersichtlich ist:

```
WHILE <Bedingung> DO
      <Anweisung1>;
      <Anweisung2>;
      <Anweisung3>;
      <Anweisung4>
END
```

Wir werden an einigen Stellen der Spracheinführung von Syntaxdiagrammen Gebrauch machen, oft aber auch eine Erläuterung an Programmbeispielen vorziehen. Der Anhang B listet alle Syntaxdiagramme auf. Er dient also als Ratgeber in Zweifelsfragen.

Abschließend sei noch erwähnt, daß Syntax-Diagramme keine Spielerei oder nur ein didaktischer Trick sind. Sie legen genau die Sprachdefinition fest, daher arbeitet auch der Compiler bei der Übersetzung ihrer Modula-Quelltexte danach. Bildlich gesprochen prüft er ab, ob der »Zug« nach dem Gleisplan fährt und erzeugt für die einzelnen »Gleise« und »Tunnel« den entsprechenden Code, den der Computer dann ausführen kann.

1.2 Übersichten

1.2.1 Die Struktur eines Modula-Programms

Wie die einführenden Beispiele gezeigt haben, besteht ein Modula-Programm aus drei Teilen:

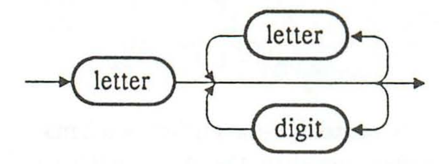
I. dem Modulkopf
II. den Importlisten
III. dem Programmblock

Zu I. (Modulkopf)

Der Modulkopf besteht aus dem Schlüsselwort `MODULE`, gefolgt von einem Modulnamen (Besonderheiten später). Ein Modulname ist ein »Bezeichner« (ebenso wie ein Variablen-Bezeichner). Für Bezeichner gelten folgende Regeln:

1. Bezeichner bestehen aus Buchstaben (»A« bis »Z« und »a« bis »z«) und Ziffern (»0« bis »9«).
2. Das ersten Zeichen muß ein Buchstabe sein.
3. Groß- und Kleinbuchstaben werden unterschieden.
4. Es gibt keine Längenbeschränkung für Bezeichner (Bezeichner können beliebig lang sein).
5. Es dürfen keine Modula-Schlüsselwörter (siehe unten) als Bezeichner verwendet werden.

Oder kürzer als Syntax-Diagramm (Buchstabe und Ziffer wird nicht weiter aufgeschlüsselt, da klar ist, was gemeint ist):



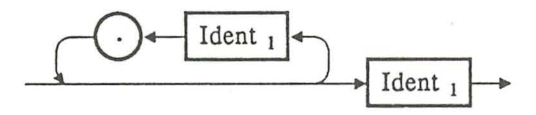
SYNTAX: "Ident"(1)

In manchen Fällen reicht die Angabe eines Bezeichners nicht aus, um ihn eindeutig zuzuordnen. Wir geben ein Beispiel: Die Prozedur `Read` findet man sowohl im Modul `InOut` als auch im Modul `Terminal` (`InOut` ist etwas komfortabler). Sollte ein Programm beide `Read`-Prozeduren brauchen, so reicht der Name `Read` im Programm nicht zur eindeutigen Identifikation aus. Man behilft sich folgendermaßen:

Die Importliste lautet einfach:

```
IMPORT InOut;
IMPORT Terminal;
```

Beide Module beinhalten eine Prozedur `Read`. Im Programm benutzt man sie dann mit `InOut.Read` bzw. `Terminal.Read`. Man spricht hier von einem »qualifizierten« Bezeichner (von engl. *qualify* = einschränken, näher bestimmen). Einem Bezeichner `b` wird dabei ein weiterer Bezeichner `a` mit einem Punkt vorangestellt zu `a.b`.



SYNTAX: "QualIdent"(2)

Beispiele für gültige Bezeichner:

```
i
xHoch2
ZweitesBeispiel
UndNunEinLangerBezeichner
y.z
x.y.z
```

Wie man am letztem Beispiel sieht, kann ein bereits qualifizierter Bezeichner `y.z` wiederum qualifiziert werden.

Keine gültigen Beispiele sind:

`2x`	(Ziffer am Anfang)
`Zweites Beispiel`	(Leerzeichen sind nicht erlaubt)
`Zweites_Beispiel`	(Sonderzeichen sind nicht erlaubt)
`Überschrift`	(Umlaute sind keine zulässigen Buchstaben)

Zu II. (Importlisten)

Importlisten haben die folgende Gestalt:

```
FROM <Modulbezeichner> IMPORT <Bezeichner1>, <Bezeichner2>...;
```

oder

```
IMPORT <Modulbezeichner>, <Modulbezeichner>...;
```

In der ersten Form (Import mit `FROM`) stehen *Bezeichner1* und *Bezeichner2* für Konstanten (siehe unten), Datentypen, Variablen oder Prozeduren (siehe unten) aus diesem Modul. Mehrere Bezeichner werden also durch Kommata getrennt; am Ende steht ein Semikolon.

Bei der zweiten Form (Import ohne »`FROM`«) werden nur die Modulnamen importiert. Will man die Bezeichner dieser Module ansprechen, müssen sie mit dem Modulnamen qualifiziert werden:

<Modulname>. <Bezeichner>

Zu III. (Programmblock)

Der Programmblock besteht aus

1. Deklarationsteil und
2. Anweisungsteil.

Im Deklarationsteil werden Programmbestandteile wie Konstanten, Datentypen und Variablen deklariert, aber auch Unterprogramme, sogenannte »Prozeduren« oder »lokale Module«.

Anders als in Pascal gibt es hier keine strenge Reihenfolge!

Die meisten Compiler akzeptieren jedoch nicht eine völlig beliebige Reihenfolge: so muß zum Beispiel eine Variable vor ihrer Benutzung deklariert sein.

Der Anweisungsteil beginnt immer mit dem Schlüsselwort `BEGIN` und endet immer mit `END` *<Modulname>*.

Zwischen `BEGIN` und `END` werden dann die Anweisungen, die das Programm später ausführen soll, in der Reihenfolge aufgelistet.

1.2.2 Reservierte Wörter der Sprache

Im folgenden geben wir eine alphabetische Liste der 40 reservierten Wörter und erklären knapp ihre Bedeutung. Hier wird natürlich noch nicht alles in voller Klarheit ausgebreitet; die Einzelheiten gehen aus den nächsten Abschnitten hervor!

`AND`	logischer Operator: »UND«
`ARRAY`	Typdeklaration: Feld
`BEGIN`	leitet den Anweisungsteil eines Moduls oder einer Prozedur ein
`BY`	dient zur Angabe der Schrittweite bei einer FOR-Schleife

`CASE`	Fallunterscheidung oder varianter Record
`CONST`	Einleitung einer Konstantendeklaration
`DEFINITION`	Einleitung eines Definitionsmoduls
`DIV`	Operator: Ganzzahlige Division
`DO`	Bestandteil einer FOR- oder WHILE-Schleife
`ELSE`	Bestandteil einer IF-Anweisung
`ELSIF`	Bestandteil einer IF-Anweisung
`END`	begrenzt einen Block, eine Prozedur oder einen Modul
`EXIT`	Abbruch einer LOOP-Schleife
`EXPORT`	leitet bei lokalen Modulen die Exportliste ein
`FOR`	Einleitung einer FOR-Schleife
`FROM`	Schlüsselwort in Importlisten
`IF`	Einleitung einer Verzweigung
`IMPLEMENTATION`	Einleitung eines Implementationsmoduls
`IMPORT`	Schlüsselwort in Importlisten
`IN`	logischer Operator für Mengen (»ist Element aus«)
`LOOP`	Einleitung einer LOOP-Schleife
`MOD`	Operator: Modulus (Rest bei der ganzzahligen Division)
`MODULE`	Einleitung eines Moduls
`NOT`	logischer Operator: »nicht«
`OF`	Bestandteil einer Felddeklaration: `ARRAY OF`...
`OR`	logischer Operator: »oder«
`POINTER`	Typdeklaration: Zeiger
`PROCEDURE`	Einleitung eines Unterprogramms (Prozedur); auch zur Typdeklaration eines Prozedur-Typs
`QUALIFIED`	Schlüsselwort in Export-Listen: `EXPORT QUALIFIED`... (qualifizierter Export: erzwingt Qualifizierung der exportierten Bezeichner)

`RECORD`	Typdeklaration: Verbund
`REPEAT`	Einleitung einer REPEAT-Schleife
`RETURN`	beendet eine Prozedur, liefert bei Funktionsprozeduren gleichzeitig den Rückgabewert
`SET`	Typdeklaration: Menge
`THEN`	Bestandteil einer IF-Anweisung
`TO`	Bestandteil einer FOR-Schleife oder Zeigerdeklaration
`TYPE`	Einleitung von Typdeklarationen
`UNTIL`	Bestandteil einer REPEAT-Schleife
`VAR`	Einleitung von Variablendeklarationen
`WHILE`	Einleitung einer WHILE-Schleife
`WITH`	dient zur Dereferenzierung von Variablen vom Typ `RECORD`

Diese Schlüsselwörter sind nicht zu verwechseln mit den Bezeichnern der Standardtypen oder Standardprozeduren von Modula, die ebenfalls alle groß geschrieben werden.

1.2.3 Übersicht über die Standard-Datentypen

Ebenso wie man aus der Mathematik natürliche Zahlen (positive und negative) und reelle Zahlen (die mit dem Komma) kennt, gibt es in Modula auch verschiedene Zahlentypen. Dazu kommen noch Zeichen, Wahrheitswerte und Mengen als einfache Datentypen:

Bezeichner	**Datentyp**	**Größe in Byte**	**Wertebereich**
INTEGER	ganze Zahl	2	$-2^{15}..2^{15}-1$
LONGINT	ganze Zahl	4	$-2^{31}..2^{31}-1$
CARDINAL	natürliche Zahl	2	$0..2^{16}-1$
LONGCARD	natürliche Zahl	4	$0..2^{32}-1$
REAL	Bruchzahl	4 oder 8	siehe unten

Bezeichner	Datentyp	Größe in Byte	Wertebereich
LONGREAL	Bruchzahl	8	siehe unten
CHAR	Zeichen	1 oder 2	CHR(0)..CHR(255)
BOOLEAN	Wahrheitswert	1 oder 2	FALSE, TRUE
BITSET	Bit-Menge	2	SET OF [0..15]

Die Größe der Datentypen und damit der Wertebereich hängt vom verwendeten Compiler ab. Unterschiede ergeben sich bei den Typen `BOOLEAN`, `CHAR`, `REAL` und `LONGREAL`:

Für `BOOLEAN` und `CHAR` reicht ein Byte, manche Compiler belegen aber zwei Bytes. Die meisten Modula-Compiler (zum Beispiel Hänisch, TDI, SPC...) implementieren `REAL` mit 4 Byte und `LONGREAL` mit 8 Byte. Megamax-Modula und MSM2 stellt nur den Typ `REAL` zur Verfügung, aber gleich mit 8 Byte. Ein 8-Byte-Real besitzt im allgemeinen eine größere Genauigkeit und einen größeren Wertebereich als mit nur 4 Byte; aber auch dies ist von der jeweiligen Implementation abhängig. Ein Megamax-`REAL` besitzt beispielsweise eine Genauigkeit von ca. 13 Stellen bei einem Wertebereich von -10^{1233} bis $+10^{1233}$.

Einige dieser Typen lassen sich unter dem Begriff »skalare Typen« zusammenfassen. Zu diesen Typen zählen die Standardtypen `INTEGER`, `LONGINT`, `CARDINAL`, `LONGCARD`, `CHAR` und `BOOLEAN` sowie die Aufzählungstypen (siehe 1.6.1). Ihnen ist gemeinsam, daß man alle ihre Werte aufzählen kann; ihre Werte sind quasi durchnumeriert. Die beiden Typen `REAL` und `LONGREAL` gehören nicht dazu. Skalare Typen zeichnen sich eben – wegen ihrer Aufzählbarkeit – durch folgende Eigenschaften aus:

- Sie besitzen genau einen Vorgänger und einen Nachfolger. Damit lassen sich die Standardprozeduren `INC` und `DEC` auf sie anwenden.
- Man kann sie für Laufvariablen in `FOR`-Schleifen und Selektoren in `CASE`-Anweisungen (Fallunterscheidung) benutzen.
- Sie lassen sich (teilweise) als Indextyp für Felder verwenden (siehe Abschnitt 1.6.3).

Aus diesen Standardtypen kann der Progammierer komplexere Datentypen, sogenannte strukturierte Typen bilden (zum Beispiel Felder). Hierzu der Abschnitt 1.6

1.2.4 Standardkonstanten in Modula

Konstante	Typ	Bedeutung
`FALSE`, `TRUE`	BOOLEAN	falsch, wahr
`NIL`	POINTER TO...	Zeiger ins »Nichts«
Zusätzlich bei Megamax-Modula:		
`MaxCard`	CARDINAL	$2^{16}-1 = 65535$
`MaxLCard`	LONGCARD	$2^{32}-1 = 4294967295$
`MaxInt`	INTEGER	$2^{15}-1 = 32767$
`MinInt`	INTEGER	$-2^{15} = -32768$
`MaxLInt`	LONGINT	$2^{31}-1 = 2147483647$
`MinLInt`	LONGINT	$-2^{31} = -2147483648$

Diese zusätzlichen Konstanten sind nicht erforderlich; die Werte lassen sich mit den Standardfunktionen `MIN` und `MAX` erhalten (s. u.).

1.2.5 Übersicht über die Standardprozeduren

In Modula gibt es nur 18 Standardprozeduren. Das sieht nach wenig aus, hat aber einerseits den Vorteil, daß man sich nicht viel merken muß, anderseits kann man ja beliebig viele Prozeduren aus Modulen importieren.

Man unterscheidet Funktionsprozeduren (kurz: Funktionen) von Prozeduren im eigentlichen Sinne. Prozeduren führen bestimmte Anweisungen aus, Funktionen liefern darüber hinaus ein Ergebnis zurück, welches man einer Variablen zuweisen kann. Beispiel:

```
ch1:= CAP(ch2);
```

`ch2` heißt dabei »Argument« der Funktion `CAP`.

`ABS(x)` *absolute value*, »Absolutbetrag«

Funktion; liefert den Absolutwert (Betrag) des Arguments. Das Ergebnis ist vom gleichem Typ wie das Argument. Zugelassen sind `INTEGER`, `LONGINT`, `REAL` und `LONGREAL`.

Beispiel: nach `i:=ABS(-3)` ist `i` gleich `3`.

`CAP(ch)` *capital letter*, »Großbuchstabe«
Funktion; Argument und Ergebnis vom Typ `CHAR`. Konvertiert einen Kleinbuchstaben in den entsprechenden Großbuchstaben.
Beispiel: nach `ch:=CAP("a")` ist `ch` gleich `"A"`

`CHR(i)` *character*, »Zeichen«
Funktion; ergibt das Zeichen mit der Ordnungszahl `i` im ASCII-Zeichensatz (siehe Anhang D). Der Ergebnis-Typ ist `CHAR`, der Argument-Typ ist `CARDINAL` oder `INTEGER`.
Beispiel: nach `ch:=CHR(65)` ist `ch` gleich `"A"`

`DEC(x)` *decrease*, »erniedrige, vermindere«
Prozedur; ersetzt `x` durch seinen Vorgänger (den nächst kleineren Wert). `x` ist eine Variable beliebigen skalaren Typs. Wenn `x` vom Typ `INTEGER` ist, entspricht die Prozedur der Anweisung `x:=x-1`.
Beispiele:
nach `c:="D";DEC(c);` ist `c` gleich `"C"`
nach `i:=5;DEC(i);` ist `i` gleich `4`.

`DEC(x,n)` *decrease*, »erniedrige, vermindere«
Prozedur; dekrementiert `x` um `n`; `n` ist dabei vom Typ `CARDINAL`. Ansonsten funktioniert `DEC(x,n)` wie ein `n`-maliger Aufruf von `DEC(x)`.
Beispiele:
nach `c:="D";DEC(c,3);` ist `c` gleich `"A"`
nach `i:=5;DEC(i,3);` ist `i` gleich `2`.

`EXCL(m,i)` *exclude*, »ausschließen«
Prozedur; entfernt `i` aus der Menge `m`. `i` ist vom skalaren Typ `T`, die Menge vom Typ `SET OF T`.
Beispiel:
nach `m:={3,4,5,6};EXCL(m,4)` ist `m` gleich `{3,5,6}`.

`FLOAT(i)` *floating point*, »Fließkomma« (reelle Zahl)
Funktion; Typ des Argumentes `i` ist standardmäßig `CARDINAL` (bei Megamax auch `LONGCARD`). `FLOAT` konvertiert `i` in die entsprechende `REAL`-Zahl.
Beispiel: nach `x:=FLOAT(5)` ist `x = 5.0`.

`HALT` *halt*, »beenden«
Diese Prozedur beendet sofort die Programmausführung. Manche Systeme geben nicht nur eine Fehlermeldung mit der Position des `HALT` aus, sondern gestatten es noch, mit einem Debugger den Programmzustand zu untersuchen. Sehr nützlich bei Fehlersuche!

`HIGH(a)` *highest index*, »höchster Index«
Funktion; `a` ist ein Feld (auch Open Array) `ARRAY OF` ... Das Ergebnis ist `CARDINAL` und liefert die Anzahl der Feldelemente minus 1. Das ist der höchste Index, wenn der kleinste 0 (Null) ist.

`INC(x)` *increase*, »erhöhen«
Prozedur; ersetzt `x` durch seinen Nachfolger (den nächst größeren Wert). `x` kann von jedem beliebigen skalaren Typ sein. Wenn `x` vom Typ `INTEGER` ist, entspricht die Prozedur der Anweisung `x:=x+1`.
Beispiele:
nach `c:="D"; INC(c);` ist `ch` gleich `"E"`
nach `i:=5; INC(i);` ist `i` gleich `6`.

`INC(x,n)` *increase*, »erhöhen«
Prozedur; erhöht `x` um `n`; `n` ist dabei vom Typ `CARDINAL`. Ansonsten funktioniert `INC(x,n)` wie ein `n`-maliger Aufruf von `INC(x)`. Beispiele:
nach `c:="D"; INC(c,3);` ist `c="G"`
nach `i:=5; INC(i,3);` ist `i` gleich `8`.

`INCL(m,i)` *include*, »einschließen«
Prozedur; fügt `i` in die Menge `m` ein. `i` ist vom skalaren Typ `T`, wenn die Menge vom Typ `SET OF T` ist.
Beispiel:
nach `m:={3,5,6}; INCL(m,4)` ist `m={3,4,5,6}`.

`MAX(T)` *maximum*, »Maximum«
Funktion; ergibt den größten Wert des Typs `T`. `T` ist `REAL`, `LONGREAL` oder ein skalarer Typ.
Beispiel: nach `i:=MAX(CARDINAL)` ist `i` gleich `65535`.

`MIN(T)` *minimum*, »Minimum«
Funktion; ergibt den kleinsten Wert des Typs `T`; wie bei `MAX` ist `T` `REAL`, `LONGREAL` oder ein skalarer Typ.
Beispiel: nach `i:=MIN(CARDINAL)` ist `i` gleich `0`.

`ODD(i)` *odd*, »ungerade«
Funktion vom Typ `BOOLEAN`. `i` ist vom Typ `INTEGER/LONGINT` oder `CARDINAL/LONGCARD`.
`ODD` liefert `TRUE`, falls `i` ungerade ist, und `FALSE`, falls `i` gerade ist.

`ORD(x)` *ordinal number*, »Ordnungszahl«
Funktion; `x` ist von einem skalaren Typ, das Ergebnis ist `CARDINAL`. `ORD(x)` liefert das x-te Element in der Wertemenge des betreffenden Types von `x`. Das erste Element eines Aufzählungstypes hat die Ordinalzahl 0 (Null).
Beispiel: nach `i:=ORD("A")` ist `i` gleich 65, da `"A"` das Zeichen mit der Nummer 65 im ASCII-Zeichensatz ist.

`SIZE(x)` *size*, »Größe«
Funktion; ergibt die Anzahl der Bytes, die die Variable `x` belegt. `x` ist eine Variable beliebigen Typs, der Ergebnistyp ist `CARDINAL` (oder `LONGCARD`, z.B. bei Megamax).

`TRUNC(x)` *truncate*, »abschneiden«
Funktion; ergibt den ganzzahligen Anteil der `REAL`-Zahl `x` als `CARDINAL` (bei Megamax: `LONGCARD`). Die Nachkommastellen werden dabei »abgeschnitten«.
Beispiel: nach `i:=TRUNC(3.9)` ist `i` gleich `3`.

Die Ausführungen über die Standardprozeduren beziehen sich auf den Megamax-Compiler. Die anderen Compiler, die wir benutzt haben, folgen im großen und ganzen dieser Liste. Es gibt aber Abweichungen:

1. Durch Spracherweiterungen: Etliche Systeme bieten über den Wirth-Standard hinausgehende Prozeduren.
2. Bei einigen Standardprozeduren sind andere Datentypen als in der obigen Liste erforderlich. Dies gilt insbesondere für die Prozeduren `DEC`, `FLOAT`, `HIGH`, `INC`, `SIZE` und `TRUNC`.

Bitte entnehmen Sie die Besonderheiten Ihrem Handbuch. Es würde den Rahmen unserer Einführung sprengen, wenn wir auf alle Einzelheiten der bekannten Compiler eingehen. Außerdem entstünde hierdurch ein schlecht lesbarer Stil. Unsere Programmbeispiele wurden alle auf dem Megamax-System getestet. Dies hat den Vorteil, daß sich alle Programme durch ein einheitliches Design auszeichnen.

Das war es schon, bleiben noch compilerspezifische Prozeduren:

Bei SPC-Modula gibt es zusätzlich für die langen Typen noch

`FLOATD(i)` Funktion; konvertiert den `INTEGER/LONGINT` Wert `i` in die entsprechende `LONGREAL`-Darstellung

`TRUNCD(x)` Funktion; wandelt `LONGREAL` `x` in eine entsprechende `LONGINT`-Zahl.

Bei einigen Compilern (mit `LONG`-Typen) gibt es zusätzlich:

`LONG(i)` Funktion; wandelt den Wert `i` vom Typ `INTEGER`, `CARDINAL`, `REAL` oder `WORD` in den entsprechende `LONG`-Typen um

`SHORT(Li)` Funktion; wandelt `LONGINT`, `LONGCARD`, `LONGREAL` oder `LONGWORD` in den entsprechende kurzen Typ um.

Bei SPC-Modula findet man diese beiden Funktionen im Modul `SYSTEM`.

Hänisch-Modula entspricht bei etlichen Prozeduren nicht dem obigen Standard. Das hat seinen Grund darin, daß `CARDINAL` und `INTEGER` wahlweise 2 Byte oder 4 Byte belegen können. Die Typen der Standardprozeduren sind entsprechend flexibel. Außerdem verfügt Hänisch-Modula noch über Prozeduren zur Stringbehandlung, die normalerweise von einem Modul `Strings` importiert werden müssen.

Wie an diesem Abschnitt zu sehen ist, gibt es leider Abweichungen je nach verwendetem Compiler. Inkompatibilitäten bereiten stets den größten Frust beim Programmieren. Fluchen Sie also nicht gleich, wenn Ihr Compiler beim Übersetzten der Quelltexte unserer Diskette etwas anmeckert. Die Programme wurden sorgfältig mit dem Megamax-Compiler der Version 3.6 entwickelt. Für die beiden ersten Kapitel wurde Wert darauf gelegt, möglich wenig Systemeigenheiten zu benutzen.

Wenn Ihren Compiler also etwas stört, so wird es an den oben genannten Differenzen liegen. Schauen Sie deshalb in Ihr Handbuch, welchen Datentyp zum Beispiel `HIGH` oder `TRUNC` liefert. Durch kleine Änderungen in diesem Bereich sollten Sie unsere Programme stets zum laufen bringen. Die Aussagen dieses Buches beziehen sich auf den Stand von Anfang 1989. In Zweifelsfällen sollten Sie also aktuelle Informationen zu Ihrem System zu Rate ziehen, da die Dinge noch im Fluß sind. Aus diesem Grunde lohnt es sich nicht, mit einer Raubkopie zu arbeiten.

Unterschiedlichkeiten einzelner Dialekte gibt es auch in anderen Sprachen. Im Gegensatz zu Basic oder Pascal ist Modula noch hoch kompatibel. Dies zeigt sich zum Beispiel daran, daß wir Module des Kapitels 5 auf einem anderen Rechner entwickelt haben.

Die genannten Unterschiede dürften sich noch weiter verringern, wenn für Modula eine Standardnorm vorliegt. Die Internationale Standardisierungs-Organisation ISO befaßt sich seit mehreren Jahren mit der Normierung von Modula-2. Es wird damit gerechnet, daß diese Norm Ende 1989 vorliegt. Der Hersteller von SPC-Modula hat bereits versprochen, sich dieser Norm anzuschließen. Es ist zu hoffen, daß andere Systeme dies auch tun werden. Bis dahin gilt: »Vive la difference«.

1.2.6 Operatoren und Begrenzer

In Modula finden folgende Sonderzeichen bzw. Symbole Verwendung:

Zeichen	Bedeutung
+	Addition; auch Vereinigung von Mengen
-	Subtraktion; auch Differenz von Mengen
*	Multiplikation; auch Schnitt von Mengen
/	Division; auch »symmetrische Differenz« von Mengen
:=	Zuweisung (lies »wird zu« oder »definitionsgemäß gleich«)
&	logisches »Und«, gleichbedeutend mit AND
~	logisches »Nicht«, gleichbedeutend mit NOT
#	ungleich
<>	wie #: ungleich
<	kleiner
<=	kleiner oder gleich
>	größer
>=	größer oder gleich
( )	Klammern (Vorrang in Ausdrücken)
[ ]	Indexklammern
{ }	Mengenklammern
(* *)	Kommentarklammern
^	Dereferenzier-Operator (für Zeiger)
, . ; .. :	Satzzeichen

1.2.7 Übersicht für Aufsteiger von Pascal

Dieser Abschnitt ist für erfahrene Pascal-Programmierer gedacht. Da Modula ein Nachfolger von Pascal ist, gibt es viele Übereinstimmungen, so daß hier nur die unterschiedlichen Sprachelemente aufgezeigt werden. Dies gibt dem Pascal-Progammierer eine schnelle Einstiegsmöglichkeit, so daß weite Teile des gesamten ersten Kapitels »diagonal« gelesen werden können. Ausnahmen bilden jedoch die Abschnitte 1.3, 1.4, 1.5.3, 1.6.7, 1.7 und 1.8, die Elemente jenseits von Pascal enthalten.

Wenn Sie sich nicht in Pascal heimisch fühlen, sollten Sie diesen Abschnitt überschlagen und das restliche erste Kapitel gründlich durcharbeiten.

Was ist neu in Modula gegenüber Pascal?

Schreibweise

Es wird streng zwischen Groß- und Kleinschreibung unterschieden. So sind `zahl`, `ZAHL` und `Zahl` verschiedene Bezeichner. Der Unterstrich »_« ist in Bezeichnern nicht zulässig (auch wenn ihn manche Compiler auf dem Atari durchgehen lassen).

Kommentare

dürfen nur in `(*...*)` eingeschlossen sein, geschweifte Klammern (`{...}`) sind für Mengen reserviert. Kommentare dürfen geschachtelt sein.

Programmstruktur

Ein Programm beginnt mit dem Schlüsselwort `MODULE` (statt `PROGRAM`), dann können Importlisten folgen.

Deklarationsteil

Im Deklarationsteil können Konstantendeklarationen, Variablendeklarationen und Prozedurdeklarationen in beliebiger Reihenfolge und an beliebiger Stelle erfolgen. Variablendeklarationen können zum Beispiel am Anfang des Programms und dann noch mehrmals mittendrin erfolgen. Das gleiche gilt für die anderen Deklarationen.

Die meisten Modula-Compiler auf dem Atari (die Single-pass-Compiler, die den Programmtext nur einmal durchlaufen) sehen jedoch eine Einschränkung vor: Bei ihnen muß jeder Bezeichner vor seiner Verwendung deklariert sein (Ausnahme: Pointer-Deklarationen). Bei diesen Compilern sind allerdings bei Prozeduren `FORWARD`-Deklarationen erlaubt. Bei der eigentlichen Definition der Prozedur muß die komplette Parameterliste wiederholt werden.

Standardbezeichner sind überall definiert. Importierte Bezeichner sind nach dem `IMPORT` definiert, wenn sie in der Importliste erscheinen; ansonsten (Import ohne `FROM`) müssen sie bei der Benutzung qualifiziert werden.

Datentypen

Neu sind `CARDINAL` (positive Ganzzahlen mit Null) und `BITSET` (entspricht `SET OF [0..15]`). Die meisten Compiler verfügen über doppelt lange Zahlentypen:

- `LONGCARD`

Ganze Zahlen von 0 bis 4294967295

- `LONGINT`

Ganze Zahlen von -2147483648 bis 2147483647

- `LONGREAL`

`REAL` mit größerer Genauigkeit/Wertebereich

Konstantendeklarationen

enthalten ihren Typ in den meisten Fällen implizit:

`c=15;`	`CARDINAL`-Konstante
`c=17B;`	oktale `CARDINAL`-Konstante ($17_{oktal} = 15_{dezimal}$)
`c=0FH;`	hexadezimale `CARDINAL`-Konstante (das erste Zeichen muß eine Ziffer sein)
`c=15D;`	`LONGCARD`-Konstante
`c=-15D;`	`LONGINT`-Konstante
`r=4.0;`	oder `r=4.` ; `REAL`-Konstante (immer mit Punkt)
`ch="A";`	oder `c='A'` ; `CHAR`-Konstante
`ch=101C;`	`CHAR`-Konstante (ASCII-Wert oktal!)
`s="Alles klar"`	oder `s='Alles klar'` String Der String darf durch beide Arten von Anführungsstrichen (einfache und doppelte) begrenzt sein. Am Anfang und Ende muß aber der gleiche stehen; der andere darf dann im String vorkommen. `s="Der 'Freak'"` oder `s='Der"Freak"'`

`m={1..3};`	Menge vom Typ `BITSET`;
	Bei anderen Mengen als `BITSET` muß der Typ mitangegeben werden.
	`TYPE VierMenge=SET OF [1..4];`
	`CONST m=VierMenge{1..3};`

Typdeklarationen

Unterschiede gibt es nur bei

a) Unterbereichstypen
b) Mengen
c) Varianten Verbunden
d) Zeiger (»Pointer«)
e) Prozedur-Typen (gibt's in Pascal nicht)
f) Dateien (»Files«)
g) Zeichenketten (»Strings«)

Zu a) Unterbereichstypen
Statt (Pascal): `Unterbereich=A..Z`
schreibt man in Modula: `Unterbereich=[A..Z]`. Der Ausdruck `[0..99]` ist ein Unterbereich des Types `CARDINAL`, hingegen ist `INTEGER[1..99]`ein Unterbereich des Types `INTEGER`.

Zu b) Mengen
Mengen dürfen nicht beliebig groß sein. Auf dem Atari sind im allgemeinen nur 16 Elemente erlaubt. Damit sind Mengen wie `SET OF CHAR` nicht möglich. Einige Compiler (TDI, Megamax) lassen aber Mengen bis 65535 Elementen zu.

Zu c) Varianten Verbunden
Entnehmen Sie die Änderungen bitte dem Abschnitt 1.6.4.

Zu d) Pointer
Statt (Pascal):
`TYPE Zeiger=^REAL` schreibt man in Modula:
`TYPE Zeiger=POINTER TO REAL`, was wesentlich lesbarer ist.

Zu e) Prozedur-Typen
Den Datentyp `PROCEDURE` gibt es in Pascal nicht. Er wird in Abschnitt 1.6.7 ausführlich beschrieben.

Zu f) Files
Es gibt in Modula keinen Standard-Datentyp `FILE`. Er wird jedoch in Modulen wie `FileSystem`, `Files` oder `Streams` realisiert.

Zu g) Strings
Den Datentyp `STRING` gibt es standardmäßig in Modula nicht (wohl in Hänisch-Modula). Er wird in Modula mit `ARRAY` *<Bereich>* `OF CHAR` realisiert. Stringprozeduren gibt es in einem Modul `Strings`. Die Stringlänge steht im Gegensatz zu Pascal nicht im vordersten Element. Vielmehr erkennt man das Ende eines Strings, der nicht alle Elemente belegt, am Abschlußzeichen `0C`.

Prozedurdeklarationen

Modula gestattet als Parameter von Prozeduren den sogenannten »offenen Feldtyp«:

```
PROCEDURE p(feld: ARRAY OF T);
```

Der Index läuft hier von `0..HIGH(feld)`. Offene Felder sind nur in Parameterlisten zulässig, ansonsten ist stets wie in Pascal der Indexbereich mit anzugeben.

Funktionen hießen in Modula »Funktionsprozeduren« und werden auch durch das Schlüsselwort `PROCEDURE` eingeleitet. In einer Funktionsprozedur erfolgt die Zuweisung des Funktionswertes durch `RETURN`. Beispiel:

```
PROCEDURE quadrat(x: REAL): REAL;
BEGIN
  RETURN x*x
END quadrat;
```

Bei dem Aufruf einer Funktionsprozedur ohne Parameter stehen Klammern:

```
i:=rechne();
```

Als Ergebnistyp einer Funktion sind nur die Standard-Datentypen erlaubt. Die meisten Compiler gestatten jedoch Erweiterungen (zum Beispiel Verbunde).

Ausdrücke

In einem Ausdruck dürfen Variablen verschiedener Typen nicht miteinander vermischt werden. Operanden verschiedenen Typs müssen gegebenenfalls mittels Typkonvertierungs-Funktionen auf den Ergebnistyp transformiert werden. Hierzu dienen die Funktionen: `TRUNC`, `FLOAT`, `INTEGER`, `CARDINAL` oder dem Modul `SYSTEM`.

Operatoren

In Modula darf man anstelle von »<>« (ungleich) auch »#« schreiben. Für die logischen Operatoren `AND` und `NOT` darf man auch »&« bzw. »~« schreiben.

Kontrollstrukturen

1. LABEL- und GOTO-Anweisungen gibt es nicht.
2. Anweisungen werden nicht mehr mit `BEGIN` und `END` geklammert. Dies ist auch nicht mehr erforderlich, da die Kontrollstrukturen selbst ein `END` erfordern.
3. Modula kennt dieselben Schleifentypen wie Pascal. Hinzu kommt noch die `LOOP`-Schleife (vgl. 1.4.1).
4. Geschachtelte IF-Anweisungen kann man in Modula mittels `ELSIF` ausdrücken:

```
IF <b1> THEN <Anweisungen1>
  ELSIF <b2> THEN <Anweisungen2>
  ELSIF <b3> THEN <Anweisungen3>
  <...>
```

```
  ELSE <Anweisungen4>
END;
```

5. Die CASE-Anweisung hat eine leicht veränderte Syntax (vgl. 1.4.2)

1.3 Vordefinierte Datentypen

Wie schon die Eingangsbeispiele zeigten, braucht ein Programm Variablen und Daten (Zahlen, Zeichen, Zeichenketten, Wahrheitswerte). Die Variablen müssen im Deklarationsteil mit Namen und Typangabe vereinbart werden. Einige Beispiele zu Variablenvereinbarungen:

```
VAR
    antwort              : CHAR;
    Vorname, Nachname    : ARRAY [0..79] OF CHAR;
    Zaehler, Nenner      : INTEGER;
    GanzeZahl            : LONGCARD;
```

Man sieht, daß nach dem Schlüsselwort VAR eine Liste von Variablenbezeichnern, die durch ein Komma getrennt werden, folgt und dann nach einem Doppelpunkt eine Typbezeichnung. Das Ende markiert ein Semikolon.

Basic-Aufsteigern mag eine Variablen-Deklaration überflüssig vorkommen. Wozu dient sie überhaupt?

- Sie legt einerseits die Namen der im Programm vorkommenden Variablen fest
- und zum anderen deren Datentyp.

Letzteres ist wichtig, da damit die interne Darstellung (wieviele Bytes belegt eine Variable?) und der Wertebereich der Variablen festgelegt wird. Die Deklaration

```
VAR i: CARDINAL;
```

beispielsweise beinhaltet, daß i nur ganzzahlige Werte mit $0 \leq i \leq 65535$ annehmen kann.

Auf diese Weise kann der Compiler beim Übersetzen und das Laufzeitsystem beim Ablauf des Programmes überprüfen, ob i innerhalb dieses Bereiches liegt.

Des weiteren werden durch die Typdeklaration auch die zulässigen Operatoren festgelegt. Der Divisionsoperator beim Typ CARDINAL heißt DIV, beim Typ REAL aber »/«.

Man sieht also: Eine Variablen-Deklaration bringt Ordnung und Übersicht, sowohl für den Leser eines Programmes als auch für den Computer.

Noch eines zu den Variablen-Bezeichnern: Sie sollten so benannt werden, daß man aus ihren Namen ihren Verwendungszweck ableiten kann. Namen der Form `i`, `x`, `y`, `z` sind allenfalls für Hilfsvariablen brauchbar.

Die nächsten Abschnitte machen genauer mit den Standard-Datentypen von Modula vertraut:

- `CARDINAL` / `LONGCARD`
- `INTEGER` / `LONGINT`
- `REAL` / `LONGREAL`
- `BOOLEAN`
- `CHAR`
- `BITSET`

1.3.1 Der Datentyp CARDINAL

Der Datentyp `CARDINAL` repräsentiert eine Teilmenge der natürlichen Zahlen und dient zum Zählen sowie für elementare Berechnungen. Genauer beinhaltet er die natürlichen Zahlen von 0 bis 65535. Dieser Wertebereich benötigt rechnerintern einen Speicherplatz von 2 Byte oder einem »Wort«.

Wir wollen uns kurz klarmachen, warum das so ist. Bekanntlich ist die kleinste Informationseinheit des Computers ein Bit, es kann die Werte 0 oder 1 annehmen. Mit zwei Bit lassen sich nun die vier Kombinationen 00, 01, 10, und 11 darstellen. Interpretiert man nun die letzte Stelle als »Einerstelle«, die davor als »Zweierstelle«, so bedeutet

11_{dual}: `1*2 + 1*1=3` genauer $1*2^1 + 1*2^0$
10_{dual}: `1*2 + 1*0=2` genauer $1*2^1 + 0*2^0$

Die obigen 4 Zahlen sind also die Dual-Darstellung der Zahlen 0, 1, 2, 3.

Weiteren Stellen ordnet man nun höhere Zweierpotenzen zu:

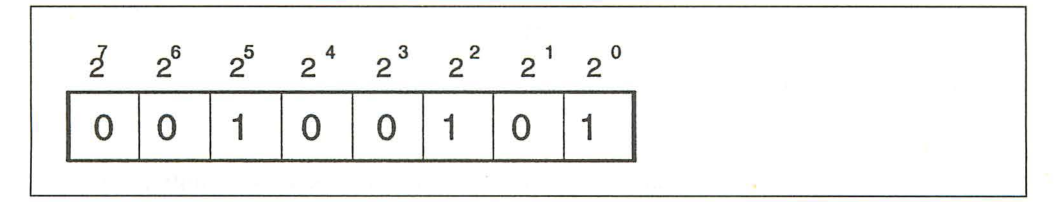


Bild 1.8: Das Byte »00100101«

Die dargestellte Folge bedeutet also

$0*2^7 + 0*2^6 + 1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0$

$= 32 + 4 + 1$

$= 37$

Acht Bit zusammen (wie oben dargestellt) nennt man ein Byte, das ist eine Speichereinheit Ihres Atari. Nebenbei bemerkt hat ihr Atari 10458576 (Modell 1040 und ST1), 2097152 (Modell ST2) oder 4194304 (Modell ST4) davon.

Die höchste Zahl, die man mit einem Byte (also 8 Bit) darstellen kann ist also

$11111111_{dual} = 255_{dezimal}$

Mit einem Byte kann man also die Zahlen von 0 bis 255 darstellen; das sind $256 = 2^8$ verschiedene Zahlen. Sie können sich so klarmachen, daß man mit 2 Byte (also 16 Bit) 2^{16} Zahlen darstellen kann (nämlich von 0 bis $2^{16}-1$), wobei die höchste Zahl $2^{16}-1 = 65535$ beträgt. Das ist der Maximalwert für `CARDINAL`-Zahlen!

Welche Operationen und Standardprozeduren lassen sich auf `CARDINAL`-Zahlen anwenden?

1. Zweistellige Operatoren:

+	Addition
-	Subtraktion
*	Multiplikation
`DIV`	ganzzahlige Division
`MOD`	Divisionsrest (Modulus)

Zur Erläuterung von `DIV` und `MOD` ein Beispiel:

   Es ist 16 = 3 * 5 + 1; also gilt:

   ```
   16 DIV 3   ergibt 5,
   16 MOD 3   ergibt 1.
   ```

   Es gelten die üblichen »Vorfahrtsregeln« wie in der Mathematik: Die Operatoren *, `DIV` und `MOD` binden stärker als + und - (»Punktrechnung geht vor Strichrechnung«). Will man es anders haben, setzt man Klammern ein.

2. Vergleichsoperatoren:

   <, >, <=, >=, =, #(oder <>)

3. Auf `CARDINAL` anwendbare Standardprozeduren sind (vergleiche Abschnitt 1.2.5): `ABS` (Ergebnis = Argument, keine Wirkung), `CHR`, `DEC`, `EXCL`, `FLOAT`, `INC`, `INCL`, `MAX`, `MIN`, `ODD`, `ORD` (Ergebnis = Argument, keine Wirkung).

Einige Beispiele zum Umgang mit CARDINAL-Zahlen

1) Die Anweisung

```
IF a < b THEN Minimum := a ELSE Minimum := b END;
```

berechnet das Minimum zweier CARDINAL-Zahlen a und b.

2) Die nächste Anweisungsfolge berechnet den ggT (größter gemeinsamer Teiler) und das kgV (kleinste gemeinsame Vielfache) zweier Zahlen a und b nach dem Euklid'schen Algorithmus:

```
VAR
    a, b     : CARDINAL;
    ggT, kgV : CARDINAL;
    rest, ab : CARDINAL;
....
ab := a * b;
REPEAT
  rest := a MOD b;
  a := b;
  b := rest
UNTIL rest = 0;
ggT := a;
kgV := ab DIV ggT;
```

Überlegen Sie sich anhand von Zahlenbeispielen, wie dieser Algorithmus funktioniert und vergleichen Sie mit dem dritten Einführungsbeispiel!

3) Das nächste Programmstück errechnet a^b (für b≧) (in Modula gibt es keinen Operator für »hoch«):

```
VAR a, b, Ergebnis : CARDINAL;
....
Ergebnis := 1;          (* Vorbereitung *)
WHILE b > 0 DO Ergebnis := Ergebnis * a; DEC(b) END;
```

Hier wird also die Potenz durch fortlaufendes Multiplizieren mit der Basis errechnet und der Exponent in jedem Schritt um 1 erniedrigt. Insgesamt sind b Multiplikationen nötig.

4) Schneller läuft der folgende raffiniertere Algorithmus:

```
VAR a, b, Ergebnis : CARDINAL;
....
```

```
Ergebnis := 1;            (* Vorbereitung *)
WHILE b > 0 DO
  IF ODD(b) THEN Ergebnis := Ergebnis * a END;
  a := a * a;
  b := b DIV 2
END;
```

Zur Erinnerung: `ODD(b)` bedeutet »b ist ungerade«. Es wird also fortlaufend die Basis quadriert, der Exponent entsprechend halbiert. Machen Sie sich die Funktionsweise beider Algorithmen am Beispiel 2^{10} klar!

5) Berechnung der Fakultät von n (mathematische Schreibweise: `n!`), also des Terms

   $n *(n-1) *(n-2) \ldots 3 *2 *1$;
   somit ist
   $4! = 4*3*2*1 = 24$

```
VAR Fakultaet, i : CARDINAL;
....
Fakultaet := 1;                (* Vorbereitung *)
FOR i := 1 TO n DO Fakultaet := Fakultaet * i END;
```

6) Das letzte Beispiel errechnet die Quersumme einer Zahl:

```
Quersumme := 0;                (* Vorbereitung *)
WHILE n > 0 DO
  Quersumme := Quersumme + n MOD 10;
  n := n DIV 10
END;
```

Folgendes sei dem Leser dringend empfohlen:

- Wenn Sie keine oder nur wenig Programmiererfahrung haben, sollten Sie sich die Funktionsweise der Beispiele anhand von konkreten Zahlen genau klarmachen.

- Wenn Sie noch keine Erfahrung im Umgang mit Modula haben, sollten Sie diese kurzen Anweisungsfolgen, die sich nicht auf der Diskette befinden, abtippen. Nehmen Sie dazu das dritte Eingangsbeispiel als »Rahmenprogramm«, laden es in den Editor und tauschen Sie die Variablenlisten und den Anweisungsteil aus! Die Importliste sowie die äußere `REPEAT...UNTIL`-Schleife zur wiederholten Durchführung kann bestehen bleiben. Die Ein- und Ausgabe muß den einzelnen Problemen angepaßt werden. Kompilieren Sie jedes Beispiel, und führen Sie es aus!

1.3.2 Der Datentyp LONGCARD

Gelegentlich reicht der Zahlenbereich von `CARDINAL` nicht aus. Zum Beispiel entstehen in unserem Programmstück zur Potenz a^b und zur Fakultät `n!` schnell Zahlen, die über 65535 wachsen. Man geht dann zum Typ `LONGCARD` über, auf dem dieselben Operatoren angewandt werden können, der aber intern in 4 Byte = 32 Bit abgespeichert wird. Nach dem oben erklärten Prinzip ist also

$11111111111111111111111111111111_{dual} = 2^{32} - 1 = 4294967295$

Das dürfte reichen, auch um die Kilometerleistung Ihres PKW abzuspeichern!

Um die Euphorie etwas zu dämpfen, muß gesagt werden, daß man in einem Ausdruck – eben wegen der unterschiedlichen Maschinen-Darstellung – Variablen verschiedener Typen wie `CARDINAL` und `LONGCARD` nicht vermischen darf.

Für `c: CARDINAL;` und `lc: LONGCARD;` ist der Ausdruck `c+lc` also nicht möglich. Wie auch? Von welchem Typ sollte das Ergebnis sein? Bei Zuweisungen lockern manche Compiler diese Kompatibilitätsforderung: sie gestatten die Zuweisung eines 2-Byte-Wertes auf einen 4-Byte-Wert. Andersherum kann es Schwierigkeiten geben: eine 4-Byte-Zahl kann zu groß sein, als daß sie in einem 2-Byte-Wert dargestellt werden kann.

Um zu verhindern, daß eine ungewollte Vermischung von 2-Byte-Typen und 4-Byte-Typen unerwartete Laufzeitfehler liefert, wird eine Anpassung der Datentypen mittels der zusätzlichen Standardfunktionen `SHORT` und `LONG` erreicht. Somit sind folgende Zuweisungen erlaubt:

`c:=SHORT(lc)` und `lc:=LONG(c)`.

Bei SPC-Modula ist eine Zuweisung in beiden Richtungen ohne Konvertierung möglich!

Ein arithmetischer Ausdruck muß auch dann einen einheitlichen Typ haben, wenn man zum Beispiel mit `CARDINAL` und `LONGCARD` gleichzeitig arbeiten will. Man muß sich für einen Typ entscheiden; Variablen oder Teilausdrücke anderen Types müssen also mit Typtransfer-Funktionen umgewandelt werden. So erhält man entweder einen `CARDINAL`-Ausdruck

`c+SHORT(lc)` oder einen `LONGCARD`-Ausdruck `LONG(c)+lc`.

Bei SPC sind `SHORT` und `LONG` aus `SYSTEM` zu importieren.

Modula verlangt also bei allen Ausdrücken, daß die beteiligten Operanden vom gleichen (bzw. kompatiblen, s.u.) Typ sind! Diese Forderung »nervt« anfangs, wenn sie zuvor in Basic oder C gearbeitet haben. Man sollte jedoch bedenken, daß dadurch der Programmierer gezwungen wird, sich über den möglichen Wertebereich eines Ausdrucks beim Programmieren Klarheit

zu verschaffen. Der Ausdruck `SHORT(lc)` verhindert also keine Überläufe beim Programmablauf.

Vorsicht ist geboten bei Ausdrücken, die den Wertebereich ihres Types überschreiten können; zum Beispiel führt folgende Sequenz zu einem Überlauf (*overflow*):

```
VAR x,a: CARDINAL;
....
a := 50000;
x := a + 30000;          (* 80000 ist mit CARDINAL nicht darstellbar!)
```

Ebenso ist

```
a := 300;
x := a * a - 30000;           (* müßte eigentlich 60000 ergeben *)
```

fehlerhaft, da der Teilausdruck `300*300` bereits einen Überlauf beinhaltet (90000 ist halt nicht durch `CARDINAL` darstellbar), obwohl das Ergebnis nur 60000 ist.

Allgemein gilt: Der Programmierer muß sicherstellen, daß alle Ausdrücke (auch Teilausdrücke) zur Laufzeit keinen Überlauf oder Unterlauf ergeben. Ansonsten kann es zu einem Programmabbruch zur Laufzeit führen. Der Compiler kann das im allgemeinen nicht merken, da er nicht im voraus feststellen kann, welche Werte die Variablen zur Laufzeit annehmen werden.

Zur Erläuterung des Umgangs mit `LONGCARD` greifen wir das 5. Beispiel des vorigen Absatzes auf und schreiben ein vollständiges Programm:

```
MODULE FakultaetBerechnung;

FROM InOut IMPORT ReadCard, WriteCard, WriteString, WriteLn, Read;

VAR
    n,i       : CARDINAL;
    fakultaet : LONGCARD;
    antwort   : CHAR;

BEGIN
  WriteString("Programm zur Berechnung von n!");
  REPEAT
    WriteLn; WriteLn;
    REPEAT
      WriteString("Geben Sie 'n' ein (0 <= n <= 12): "); ReadCard(n);
```

```
    IF n > 12 THEN WriteString("Zu groß, Wiederholung!"); WriteLn END;
   UNTIL n <= 12;
   fakultaet:=1D;
   FOR i:=1 TO n DO fakultaet := fakultaet * LONG(i) END;
   WriteString("Es gilt: ");
   WriteCard(n,1);
   WriteString("! = "); WriteCard(fakultaet,1);
   WriteLn; WriteLn;
   WriteString("Wünschen Sie noch eine Berechnung (j/n)? ");
   Read(antwort);
   antwort := CAP(antwort);              (* Umwandlung in Großbuchstaben *)
  UNTIL antwort = "N";
END FakultaetBerechnung.
```

Sie können nun auch `12! = 479 001 600` (statt nur `8! = 40320` bei der `CARDINAL`-Version) eingeben. Wenn man noch größere Zahlen braucht, geht man zum Datentyp `REAL` über.

Das Programm zeigt einiges Neue:

1. Damit keine Zahlen über 12 vorkommen (`13! >MAX(LONGCARD)`), werden solche Eingaben in der `REPEAT`-Schleife zurückgewiesen.

2. Zur Initialisierung braucht man die `LONGCARD`-Konstante Eins. Sie heißt »`1D`«(D: *double precision* =»doppelte Genauigkeit«). Bei Megamax ist auch »`1L`«erlaubt.

3. `LONGCARD`-Zahlen können mittels `InOut.WriteCard` ausgegeben werden.

1.3.3 Die Datentypen INTEGER / LONGINT

Diese beiden Typen repräsentieren ganze Zahlen. Der Unterschied zu `CARDINAL` besteht darin, daß auch negative Zahlen mit eingeschlossen sind. Eine `INTEGER`-Variable belegt 2 Byte. Der Wertebereich von 65536 verschiedenen Werten (vergleiche Typ `CARDINAL`) verteilt sich auf den positiven und negativen Bereich: er beträgt deshalb -32768 bis 32767 (bei der positiven Hälfte wird ein Wert für die Null benutzt, darum ist sie um eins kleiner). Die positiven Zahlen (einschließlich Null) werden intern wie `CARDINAL`-Zahlen gespeichert; das höchstwertige Bit (das mit der Nummer 15, also das ganz linke) ist dabei Null. Beträgt es Eins, so handelt es sich um eine negative Zahl.

Die 32768 Zahlen von 0 bis 32767 haben die Darstellung

`0xxxxxxxxxxxxxxx,`	beispielsweise
`0000000000000001`	entspricht 1 und
`0111111111111111`	entspricht 32767.

Die 32768 negativen Zahlen von -32768 bis -1 haben die Darstellung

`1xxxxxxxxxxxxxxx,`	beispielsweise
`1111111111111111`	entspricht -1 und
`1000000000000000`	entspricht -32768.

Analog dazu existiert ein Typ `LONGINT`, in dem ganze Zahlen in 4 Byte abgespeichert werden. Er umfaßt daher den Wertebereich von -2147483648 ($=-2^{31}$) bis 2147483647 ($=2^{31}-1$)

Auf `INTEGER` / `LONGINT`-Variablen sind die gleichen Operatoren wie auf `CARDINAL`-Variablen erklärt. Das gilt auch im großen und ganzen für die Standardfunktionen (vergleiche dort).

`INTEGER`- und `LONGINT`-Variablen dürfen in einem Ausdruck nicht miteinander vermischt werden; SPC-Modula erlaubt allerdings ihre gegenseitige Zuweisung. Bei Megamax-Modula sind die Typtransfer-Funktionen `LONG` und `SHORT` zu verwenden.

Ebensowenig ist es erlaubt, in einem Ausdruck `INTEGER`- und `CARDINAL`-Typen zu mischen. Das leuchtet ein, wenn man die besondere interne Darstellung der negativen Zahlen berücksichtigt. Nach der Deklaration

```
VAR
    c1, c2: CARDINAL;
    i1, i2: INTEGER;
```

sind folgende Anweisungen nicht erlaubt:

```
c2 := c1 + i1;
i2 := i1 DIV c1;
```

Hier besteht die Notwendigkeit, einen der beiden Typen `INTEGER` und `CARDINAL` innerhalb des Ausdruckes in den anderen umzuwandeln. Man kann sehr einfach Ausdrücke in einen anderen Typ umwandeln, indem man den Typ, den der Ausdruck erhalten soll, wie eine Funktion davor schreibt:

```
c2 := c1 + CARDINAL(i1);
i2 := i1 DIV INTEGER(c1);
```

Ein solcher »Typtransfer« läßt sich auch für alle anderen Typen verallgemeinern:

```
<NeuerTyp>(<Ausdruck>)
```

Der Ausdruck erhält damit den Typ *<NeuerTyp>*.

Allgemein läßt sich jeder Ausdruck mit so einem Typtransfer in einen anderen Typ verwandeln.

Modula erlaubt es jedoch, `INTEGER`-Ausdrücke `CARDINAL`-Variablen zuzuweisen und umgekehrt; man sagt, `INTEGER` und `CARDINAL` sind untereinander »zuweisungskompatible« Datentypen. Das gleiche gilt für `LONGINT` / `LONGCARD`. Zur Verdeutlichung folgendes Beispiel: nach der Deklaration

```
VAR
    c: CARDINAL; lc: LONGCARD;
    i: INTEGER;  li: LONGINT;
```

sind folgende Zuweisungen korrekt:

```
i  := c;
lc := li;
```

Es bleibt noch zu sagen, daß man `INTEGER` / `LONGINT`-Variablen nur dann einsetzen sollte, wenn ausdrücklich negative Werte vorkommen. Ansonsten ist `CARDINAL` / `LONGCARD` vorzuziehen.

Ganzzahlen-Arithmetik läuft deutlich schneller als `REAL`-Arithmetik (`REAL`: gebrochene Zahlen, siehe nächster Abschnitt). Viele Problemstellungen, deren Lösungen auf den ersten Blick `REAL`-Arithmetik verlangt, kann man mit Ganzzahlen bearbeiten. Hier ein Beispiel:

Vielleicht haben Sie sich schon gefragt, wie ein Computer Kreise zeichnet. Hierzu müssen die Koordinaten eines jeden Kreispunktes errechnet und dann gezeichnet werden. Man denkt bei der Berechnung unwillkürlich an *Pythagoras;* die Kreisgleichung für einen Kreis mit Mittelpunkt (0,0)

$$x^2 + y^2 = r^2$$

kann man nach y auflösen. Dazu braucht man aber die Wurzelfunktion, die selbstverständlich für ganze Zahlen nicht definiert ist (für `REAL`-Variablen gibt es natürlich eine...). Eine Andere Möglichkeit wäre der Einsatz von Sinus- und Kosinus-Funktion:

$$x=r*\cos(\alpha)$$

und

$$y=r*\sin(\alpha)$$

aber hier kommt man erst recht nicht mit ganzen Zahlen aus.

Da die Bildschirmkoordinaten, an denen die Kreispunkte gezeichnet werden, ganzzahlig sind, müßten wir die berechneten reellen Zahlen ohnehin wieder auf ganze Zahlen runden. Versuchen wir also doch lieber direkt eine Ganzzahlen-Lösung:

Zunächst kann man, wenn man einen Kreispunkt `k = (x,y)` errechnet hat, wegen der eingezeichneten Symmetrieachsen sieben weitere Punkte `(y,x)`, `(y,-x)`, `(x,-y)`, `(-x,-y)`, `(-y,-x)`, `(-y,x)`, `(-x,y)` einzeichnen. Das heißt, man braucht nur die Punkte des Achtelkreises von `A` bis `B` zu ermitteln. Den Punkt A kennt man: `A=(0,r)`, wenn `r` der Radius des Kreises ist.

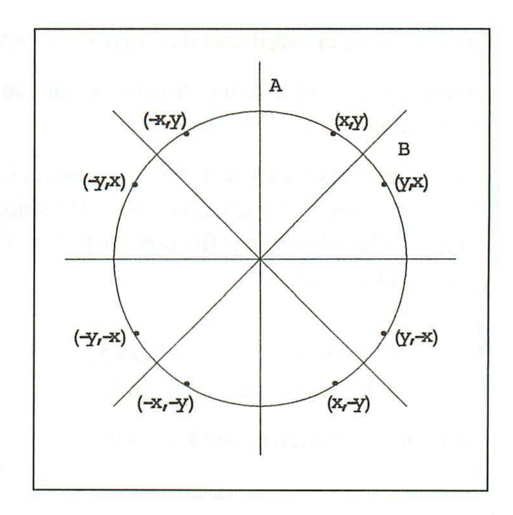


Bild 1.9: Die Achtpunkte-Symmetrie eines Kreises

Hat man aber einen Punkt `(x,y)` auf dem Kreisbogen `(AB)`, kann man den im Uhrzeigersinn nächsten nach folgendem Algorithmus (*Bresenham* 1977) ermitteln:

In Frage kommt nur der Bildschirmpunkt `P(x+1,y)`(rechts daneben) oder `Q(x+1, y-1)` (rechts unterhalb). Je nachdem, welcher Punkt vom nächsten wirklichen Kreispunkt weniger weit entfernt ist, wählt man `P` oder `Q`.

Das Quadrat der Entfernung `OP` beträgt:

$|OP|^2=(x+1)^2+y^2$

also ist

$dP=(x+1)^2+y^2-r^2$

ein Maß für die Abweichung des Punktes `P` vom Kreisrand. Es gilt: $dP \geq 0$, da `P` im Normalfall außerhalb des Kreises liegt, im Idealfall auf dem Kreis.

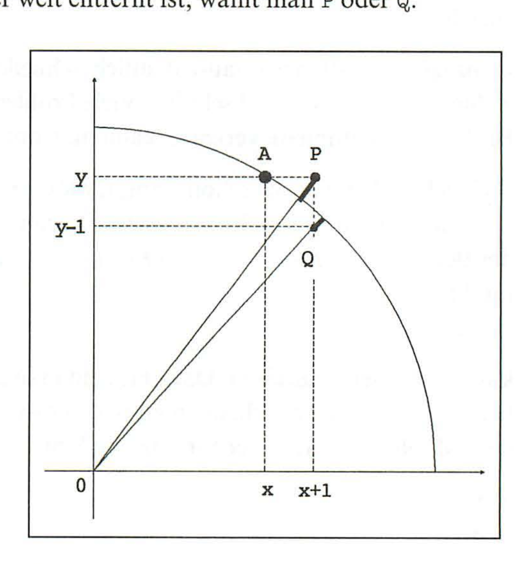


Bild 1.10: Mögl. Nachbarpunkte zum Kreispunkt A

Analog ist

$dQ=r^2-((x+1)^2+(y-1)^2-r^2)$

ein Maß für die Abweichung des Punktes `Q` vom Kreisrand. Es gilt: $dQ \geq 0$.

Die Entscheidungsregel lautet also:

ist $dP>dQ$, so wähle `Q` als nächsten Punkt, sonst `P`.

Wir formen noch etwas um:

$$
\begin{aligned}
& dP>dQ \\
\Leftrightarrow\ & dP-dQ>0 \\
\Leftrightarrow\ & 2((x+1)^2+y^2-y-r^2)+1>0 \\
\Leftrightarrow\ & (x+1)^2+y^2-y-r^2>-0.5
\end{aligned}
$$

Da hier nur ganze Zahlen vorkommen, ist diese Ungleichung äquivalent zu der Beziehung:

$$(x+1)^2+y^2-y-r^2 \geq 0$$

In Modula ausgedrückt sieht der Übergang vom Punkt (x, y) nach (x+1, y) oder (x+1, y-1) dann so aus:

```
INC(x);
IF x*x + y*y - y - r*r >= 0 THEN DEC(y) END;
```

Das war schon alles!

Der Ausdruck `r*r` ist konstant, deshalb berechnen wir ihn vor der Schleife

```
rHoch2:=r*r.
```

Dies ist ein allgemeines Prinzip zur Geschwindigkeitsoptimierung: Die Wiederholung von identischen Ausdrücken in einer Schleife sollte vermieden werden!

Da noch keine Grafik eingeführt ist (kommt in Kapitel 4) benutzen wir zu Testzwecken den Textbildschirm. Die Positionierung des Cursors an die Spalte x und die Zeile y erreicht man mit `GotoXY(x, y)` aus dem Modul `Terminal`. Da wir einen Textbildschirm mit 80 Spalten zu je 25 Zeilen haben, gilt

$0 \leq x \leq 79$ und $0 \leq y \leq 24$.

Für jeden Punkt malen wir einen Stern »*«. Der Kreis wird stark in der Vertikalen gestreckt, da auf dem Textbildschirm die Zeichenhöhe größer als die Breite ist.

Falls Sie ein anderes System als Megamax oder MSM2 benutzen, ist die Prozedur `GotoXY` nicht im Modul `Terminal` oder `InOut` enthalten, da dies nicht von *N. Wirth* vorgesehen wurde. Schreiben Sie sich einfach selber eine Prozedur `GotoXY`! Man fügt dazu nach der Variablendeklaration (vor dem `BEGIN` also) folgendes ein:

```
PROCEDURE GotoXY(x,y: INTEGER);
BEGIN
  Write(33C); Write("Y"); Write(CHR(32+x)); Write(CHR(32+y))
END GotoXY;
```

Was hier im einzelnen passiert, bleibt vorerst noch geheimnisvoll, wird aber in späteren Abschnitten erklärt. Kompilieren Sie das komplette Programm selbst und lassen Sie es laufen.

```
MODULE Kreis; (* Kreisdarstellung nach BRESENHAM 1977 *)

FROM Terminal IMPORT Read, Write, GotoXY;

VAR
    x, y, xMitte, yMitte, r, rHoch2 : INTEGER;
    taste                           : CHAR;

BEGIN
  xMitte := 40; yMitte := 12; r := 8;
  x := 0; y := r; rHoch2 := r * r;
  REPEAT
    GotoXY(xMitte+x, yMitte+y); Write("*");
    GotoXY(xMitte+y, yMitte+x); Write("*");
    GotoXY(xMitte+y, yMitte-x); Write("*");
    GotoXY(xMitte+x, yMitte-y); Write("*");
    GotoXY(xMitte-x, yMitte-y); Write("*");
    GotoXY(xMitte-y, yMitte-x); Write("*");
    GotoXY(xMitte-y, yMitte+x); Write("*");
    GotoXY(xMitte-x, yMitte+y); Write("*");
    INC(x);
    IF x*x + y*y - y - rHoch2 >= 0 THEN DEC(y) END
  UNTIL x >= y;
  GotoXY(1,20); Read(taste)
END Kreis.
```

Das Programm endet mit `Read(taste)`. Dort wartet der Rechner auf einen beliebigen Tastendruck. Das hat nur den Sinn, daß das Programm an der Stelle nicht sofort beendet wird und das Ergebnis auf dem Bildschirm sichtbar bleibt. Manche Modula-Systeme räumen nämlich nach Beendigung eines Programms sofort den Bildschirm auf, ohne daß man Gelegenheit hat, sich die Ausgaben des Programmes anzusehen.

Selbstverständlich existieren für das Zeichnen eines Kreises fertige Grafik-Routinen in der GEM-Bibliothek (vgl. Kapitel 4). Aber es empfiehlt sich dennoch zu wissen, wie so etwas gemacht wird. Auch das Zeichnen von Linien mit beliebiger »reeller« Steigung führt man auf Ganzzahl-Arithmetik zurück. Bildpunkt-Koordinaten sind eben nur ganze Zahlen.

1.3.4 Die Datentypen REAL / LONGREAL

Diese Datentypen repräsentieren eine Teilmenge der gebrochenen Zahlen. Wir sprechen hier absichtlich nicht von reellen Zahlen, da irrationale Zahlen wegen ihrer unendlichen Anzahl von Nachkommastellen nicht von einer Maschine dargestellt werden können.

`REAL`-Variablen werden bei Hänisch-, SPC- und TDI-Modula mit 4 Byte dargestellt. Daneben gibt es den Typ `LONGREAL`, er umfaßt 8 Byte. In Megamax-Modula haben `REAL`-Variablen direkt 8 Byte, den Typ `LONGREAL` gibt es hier nicht. Viele Compiler (SPC, TDI) speichern `REAL`-Variablen in einem genormten Format, dem »IEEE-Single-Precision-Format«:

Speicherbedarf: 4 Byte
Wertebereich: $-3.3 * 10^{38}$ bis $3.3 * 10^{38}$

Ebenso wird dort für `LONGREAL` mit 8-Byte das IEEE-Double-Precision-Format verwendet:

Speicherbedarf: 8 Byte
Wertebereich: $-1.79 * 10^{-308}$ bis $1.79 * 10^{308}$

Megamax benutzt ein eigenes Format, das eine schnellere Arithmetik erlaubt. Hier reicht der Wertebereich für die 8 Byte-Real bis 10^{1233}.

Alle diese Wertebereiche sollten also ausreichen, um auch größere Zahlen zu bearbeiten, wie sie zum Beispiel bei der Führung Ihres Girokontos vorkommen!

Das Abspeichern von `REAL`-Variablen ist wesentlich komplizierter als bei den Ganzzahl-Variablen und variiert auch zwischen den verschiedenen Formaten. Allen Formaten ist aber folgendes gemeinsam:

- Es wird der Exponent der Zahl zur Basis 2 gespeichert.
- Die eigentliche Ziffernfolge der Zahl, die *Mantisse*, wird auch im Dualsystem dargestellt.
- Man richtet den Exponenten beim Abspeichern so ein, daß die Mantisse `0,1xx...xx` lautet. Dies ist für alle Zahlen außer Null möglich; Null wird gesondert gekennzeichnet.
- Außerdem müssen noch das Vorzeichen der Mantisse und des Exponenten verwaltet werden.

All das braucht Sie als Modula-Programmierer nicht zu kümmern, der Computer erledigt es für Sie. Wir erwähnen es nur, um ein gewisses »Problembewußtsein« zu schaffen: Es ist klar, daß eine `REAL`-Arithmetik wesentlich langsamer als eine Ganzzahl-Arithmetik abläuft. Daher sollte man Ganzzahlen (`CARDINAL`, `INTEGER`; auch `LONGCARD` oder `LONGINT`) nutzen, wenn nicht unbedingt gebrochene oder sehr große Zahlen benötigt werden. Außerdem können we-

gen der endlichen Stellenzahl bei Real-Variablen in Berechnungen Rundungsfehler auftreten; bei Ganzzahl-Arithmetik kommt so etwas nicht vor.

Nach diesen Vorbemerkungen einige Beispielprogramme, die das Arbeiten mit gebrochenen Zahlen zeigen. Hierzu ist noch zu sagen, daß auf `REAL`-Variablen die gleichen Operatoren wie auf `CARDINAL`- und `INTEGER`-Variablen anwendbar sind mit der Ausnahme von `MOD` und `DIV`. Statt dessen gibt es das Divisions-Zeichen »/«. Der Ausdruck `5.0/2.0` ergibt `2.5`.

```
MODULE Kugel;

FROM InOut IMPORT ReadReal, WriteReal, Read, WriteString, WriteLn;

CONST Pi = 3.141592654;

VAR r, Volumen, Oberflaeche : REAL;

BEGIN
  WriteString("Berechnung des Kugelvolumens und der Oberfläche");
  WriteLn;
  REPEAT
    WriteLn; WriteString("Geben Sie den Radius in Metern ein (0 = Ende): ");
    ReadReal(r); WriteLn;
    Volumen     := 4.0 / 3.0 * Pi * r * r * r;
    Oberflaeche := 4.0 * Pi * r * r;
    WriteString("Das Kugelvolumen beträgt: ");
    WriteReal(Volumen,1,3); WriteString(" Kubikmeter,"); WriteLn;
    WriteString("Die Oberfläche beträgt:   ");
    WriteReal(Oberflaeche,1,2); WriteString(" Quadratmeter."); WriteLn;
  UNTIL r = 0.0
END Kugel.
```

Wenn Ihr Compiler die Prozeduren `WriteReal` und `ReadReal` nicht im Modul `InOut` findet, so werden sie im Modul `RealInOut` bereitgestellt. Die Importliste muß dann heißen:

```
FROM RealInOut IMPORT ReadReal, WriteReal;
FROM InOut IMPORT Read, WriteString, WriteLn;
```

Was ist neu an diesem Programm?

Zunächst haben wir einmal eine Konstanten-Deklaration, die mit `CONST` eingeleitet wird. Dem Bezeichner Pi wird der `REAL`-Wert `3.141592654` zugeordnet. Sinn einer Konstantendeklaration ist, daß der Compiler überall dort, wo im Quelltext das Symbol `Pi` auftritt, dieses Symbol durch `3.141592654` ersetzt.

Gehen wir noch kurz auf die Berechnung des Volumens ein. In der Formelsammlung finden sie für das Kugelvolumen `V` die Formel

$V = 4/3\ \pi\ r^3$

In Modula gibt es aber keinen Potenzoperator für »Hoch«, daher multiplizieren wir den Radius dreimal. Man darf aber nicht einfach

```
volumen := 4/3 *Pi*r*r*r; (*falsch!*)
```

programmieren, da 4 und 3 als `INTEGER`- oder `CARDINAL`-Konstanten erkannt werden. Eine `REAL`-Konstante muß unbedingt einen Dezimalpunkt ».« enthalten. Also `4.0` statt `4`, ansonsten meldet der Compiler einen Typ-Konflikt, da

1. der Operator / nicht auf Ganzzahlen anwendbar ist und
2. keine Ganzzahlen und `REAL`-Variablen in einem Ausdruck vermischt werden dürfen.

Im nächsten Beispiel sollen beliebig viele reelle Zahlen eingelesen werden. Es erfolgt als Ausgabe der Summe des Maximums und Minimums des Mittelwertes. Um letzteren zu errechnen, müssen wir die Eingaben zählen. Hierzu nimmt man selbstverständlich eine `CARDINAL`-Variable; wir nennen sie `anzahl`. Der Mittelwert berechnet sich bekanntlich als Quotient aus Summe durch Anzahl.Wir können aber nicht einfach schreiben:

```
MittelWert := summe / anzahl; (*falsch!*)
```

da wir dabei einen `REAL`-Ausdruck durch einen `CARDINAL`-Ausdruck dividieren würden. Deshalb muß man `anzahl` nach `REAL` konvertieren. Das geschieht mittels der Funktion `FLOAT`:

```
MittelWert := summe / FLOAT(anzahl);
```

Das ganze Programm sieht dann folgendermaßen aus:

```
MODULE MittelWertBerechnung;

FROM InOut IMPORT ReadReal, WriteReal, Read, Write,
                  WriteCard, WriteString, WriteLn;

VAR
    zahl, summe, Min, Max, MittelWert : REAL;
    anzahl                            : CARDINAL;
    taste                             : CHAR;

BEGIN
  WriteString("Mittelwertberechnung"); WriteLn; WriteLn;
  WriteString("Geben Sie fortlaufend Zahlen ein, Beenden mit 0 !");
```

```
    WriteLn; WriteLn;
    summe := 0.0; Min := MAX(REAL); Max := MIN(REAL);
    anzahl := 0;
    REPEAT
      WriteString("Zahl: "); ReadReal(zahl); WriteLn;
      IF zahl # 0.0 THEN
        INC(anzahl);
        summe := summe + zahl;
        IF zahl > Max THEN Max := zahl END;
        IF zahl < Min THEN Min := zahl END;
      END
    UNTIL zahl = 0.0;
    MittelWert := summe / FLOAT(anzahl);
    WriteString("Sie gaben insgesamt ");
    WriteCard(anzahl,1); WriteString(" Zahlen ein."); WriteLn;
    WriteString("Die Summe beträgt ");
    WriteReal(summe,1,4); Write("."); WriteLn;
    WriteString("Der Mittelwert beträgt ");
    WriteReal(MittelWert,1,4); Write("."); WriteLn;
    WriteString("Das Maximum beträgt ");
    WriteReal(Max,1,4); Write("."); WriteLn;
    WriteString("Das Minimum beträgt ");
    WriteReal(Min,1,4); Write("."); WriteLn;
    WriteLn; WriteString("Bitte Taste drücken! ");
    Read(taste)
  END MittelWertBerechnung.
```

Die drei folgenden Beispiele sollen Sie ein wenig mit der Problematik von Rundungsfehlern vertraut machen.

Wenn `x = 1.0` ist, so ist `1.0 + x` sicherlich größer als `1.0`. Das stimmt auch noch, wenn man `x` fortlaufend halbiert. Also dürfte die WHILE-Schleife im folgenden Programm nicht abbrechen:

```
MODULE RundungsDemo1;

FROM InOut IMPORT WriteReal, Read;

VAR
    x     : REAL;
    taste : CHAR;
```

```
BEGIN
  x := 1.0;
  WHILE 1.0 + x > 1.0 DO x := x/2.0 END;
  WriteReal(x,20,18);
  Read(taste)
END RundungsDemo1.
```

Tatsächlich erfolgt aber ein Abbruch, es wird beim Megamax-Compiler `1.7763568394002E-15` ausgegeben (das bedeutet ca. $1.776 * 10^{-15}$, also eine sehr kleine Zahl). Wie kommt es dazu? Offensichtlich bleibt eine Addition von Zahlen mit zu starken Größenunterschieden wirkungslos! Unterschreitet `x` eine bestimmte Größe, so wird

```
1.0 + x = 1.00000000000000xxxxxx
```

und die weiteren Stellen werden wegen der endlichen Stellenzahl des Rechners abgeschnitten; für den Rechner ist die Zahl gleich `1.0`.

Aus dem gleichem Grunde sollte man Abbruchsbedingung für Schleifen in der Form

```
REPEAT <...> UNTIL x = y;  oder
WHILE x = y DO <...> END;
```

mit `VAR x,y: REAL` vermeiden, da sich `x` und `y` um einen zu kleinen Betrag voneinander unterscheiden können. Statt dessen sollte man Abbruchsbedingungen der Form

`ABS(x - y) < 1.0E- 10` oder `x <= y` bzw. `x >= y`

verwenden. Hierzu als Beispiel:

```
x := 0.1;
summe := 0.0
REPEAT
  summe := summe + x
UNTIL summe = 10.0
```

Diese Schleife sollte eigentlich nach 100 Additionen abbrechen. Tut sie aber nicht! Denn intern kann der Rechner `0.1` nicht »glatt« als Dualzahl darstellen; dadurch wird die 100malige Addition nicht genau 10.

Das folgende Programm terminiert nur nach Tastendruck; hierzu benutzen wir die Prozedur `KeyPressed` aus dem Modul `InOut`, die `TRUE` liefert, wenn man eine Taste drückt. Wenn Ihr System `KeyPressed` nicht bereitstellt, schreiben Sie es sich selbst:

```
PROCEDURE KeyPressed: BOOLEAN;
VAR taste: CHAR;
BEGIN
  BusyRead(taste);
  RETURN taste # 0C
END KeyPressed;
```

Fügen sie diese Zeilen vor dem `BEGIN` ein. Die Importliste lautet nun:

```
FROM InOut IMPORT WriteReal;
FROM Terminal IMPORT BusyRead;
```

```
MODULE RundungsDemo2;

FROM InOut IMPORT WriteReal, KeyPressed, WriteString, Read;

VAR x, summe : REAL;
    Taste    : CHAR;

BEGIN
  x := 0.1;
  summe := 0.0;
  WriteString("Bitte nach einiger Zeit eine Taste drücken! ");
  REPEAT
    summe := summe + x;
  UNTIL (summe = 10.0) OR KeyPressed();
  WriteReal(summe,15,13);
  WHILE KeyPressed() DO Read(taste) END; Read(taste)
END RundungsDemo2.
```

Lassen Sie das Programm einige Sekunden laufen, drücken sie dann eine Taste. Sie werden sich wundern, wie groß `summe` inzwischen geworden ist.

Wenn sie das Gleichheitszeichen in der Abbruchsbedingung durch »>=« ersetzen, funktioniert alles ordnungsgemäß. Ausprobieren!

Es kommt noch toller!

Vielleicht haben Sie einen guten Mathematikunterricht genossen und können sich noch dunkel daran erinnern, daß für den Kreisumfang U bei einem Radius r. $U = 2\pi r$ gilt. »π« ist demnach der halbe Umfang eines Einheitskreises (Radius = 1). Um diesen Wert zu ermitteln, zeichnet man seit *Archimedes* (287–212 v.Chr.) regelmäßige Vielecke in einen Kreis und berechnet deren Umfang. Beginnt man mit einem Dreieck, so hat es die Seitenlänge

$s_3 = |AB| = \sqrt{3}$

Es sei x = ME.
Aus dem Dreieck DCA folgt:
$s_6{}^2 = 2\,(1-x)$ (Höhensatz, DC=2, MC=1)
Aus dem Dreieck MEA folgt wegen MA=1:
$x^2 = 1 - (s_3/2)^2$
Also ist
$s_6 = \sqrt{2-\sqrt{4-s_3{}^2}}$
Wir haben bei der Herleitung keinen Gebrauch von der speziellen Eckenzahl 3 bzw. 6 gemacht. Also gilt allgemein bei der Eckenzahl eines einbeschriebenen n-Ecks:

$$s_{2n} = \sqrt{2-\sqrt{4-s_n{}^2}}$$

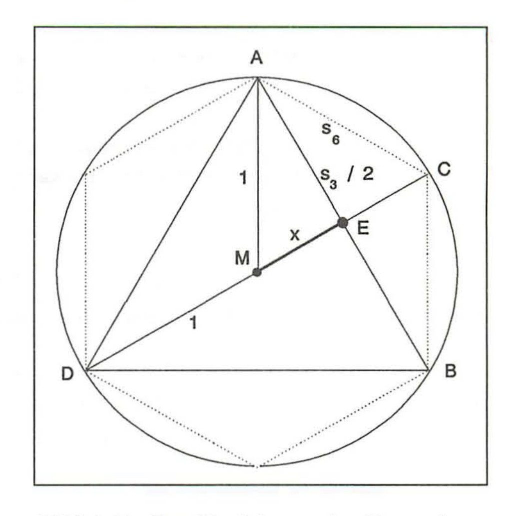


Bild 1.11: Zur Herleitung der Formel »Pi nach Archimedes«

Die Wurzelfunktion `sqrt` importiert man wie alle anderen wichtigen Funktionen für `REAL`-Variablen aus dem Modul `MathLib0` oder `MathLib`, je nach System.

```
MODULE PiNachArchimedes;

FROM InOut    IMPORT WriteReal, WriteCard, WriteString, WriteLn, Read,
                     RedirectOutput, CloseOutput;
FROM MathLib0 IMPORT sqrt;

VAR
    s         : REAL;
    n, ende   : LONGCARD;
    taste     : CHAR;

BEGIN
  n := 3;
  s:=sqrt(3.0);
  ende := MAX(LONGCARD) DIV 2D;
  Writeln; WriteString("Ausgabe auf Drucker (j/n)? "); Read(taste);
  IF CAP(taste) = "J" THEN RedirectOutput(PRN:",FALSE) END;
  WriteString("Berechnung von Pi");
  WriteLn; WriteLn;
  WHILE n < ende DO             (* aufhören, sonst Überlauf bei der Mult. mit 2 *)
    n := n * 2D;
    s := sqrt( 2.0 - sqrt( 4.0 - s*s ));
    WriteString("Eckenzahl: "); WriteCard(n,10);
```

```
    WriteString(", Näherungswert für Pi: ");
    WriteReal(s * FLOAT(n DIV 2D),15,13); WriteLn
  END;
  IF CAP(taste) = "J" THEN CloseOutput END;        (* Drucker-Ausgabe beenden *)
  Writeln; WriteString("Taste drücken! "); Read(taste);
END PiNachArchimedes.
```

Die Prozedur `RedirectOutput` aus `InOut` bewirkt die Umleitung der Ausgabe. Wir wollen den Drucker (»`PRN:`«) als Ausgabegerät ansprechen.

Zunächst sieht alles ganz gut aus, die Werte nähern sich π(=3,141592653589793...) Bei der Eckenzahl 12288 wird der Wert 3,14159265332... erreicht. Aber dann wird es drastisch schlechter, am Ende sogar =0 ! Das sollte man ausprobieren.

An der Formel

$$s_{2n} = \sqrt{2-\sqrt{4-s_n^2}}$$

erkennt man, daß bei sehr hoher Eckenzahl n der Wert s_n^2 so klein werden kann, daß der Computer $4-s_n^2$ zu 4 rundet, woraus sich für $s_{2n} = 0$ ergibt. Man spricht hier von einer »Subtraktions-Katastrophe«. Die Differenz zweier sehr nahe beieinander liegenden Zahlen kann der Computer nicht mehr von Null unterscheiden! Wie läßt sich so etwas vermeiden?

Man macht folgende geschickte mathematische Umformung. Der Term

$s_{2n} = \sqrt{2-\sqrt{4-s_n^2}}$ wird mit $\sqrt{2+\sqrt{4-s_n^2}}$ erweitert.

Es ergibt sich

$$s_{2n} = s_n / \sqrt{2+\sqrt{4-s_n^2}}$$

Hier tritt keine Subtraktionskatastrophe ein. Ändert man das Programm entsprechend ab, so erhält man (Megamax, 8-Byte-REAL's):

```
Eckenzahl:3221225472, Näherungswert für Pi: 3.1415926535899
```

Ein entsprechend abgeändertes Programm befindet sich auf der Diskette.

Diese Beispiele sollten Sie kritisch für den allzu sorglosen Umgang mit `REAL`-Variablen machen. Es sei betont, daß Rundungsfehler systemimmanent sind. Sie sind begründet in der computerspezifischen Darstellung von reellen Zahlen. Es handelt sich also nicht etwa um eine Unzulänglichkeit von Modula!

Neben der Subtraktion ist auch die Division nicht unproblematisch. Dividiert man durch eine Zahl, die sehr nahe bei Null liegt, kann es zu Überläufen kommen.

Das letzte Demoprogramm zu REAL's ist eher als »Gag« gedacht. Es zeigt eine näherungsweise Bestimmung von π mittels eines Zufallsgenerators.

Stellen Sie sich ein Quadrat von 1 m Kantenlänge vor, in das ein Viertelkreis (Radius 1 m) eingezeichnet ist. Nun regnet es, und die Viertelkreisfläche wie auch das gesamte Quadrat wird gleichmäßig naß. Wir zählen die Gesamtzahl der Regentropfen t und die Zahl derjenigen Tropfen k, die in den Viertelkreis fallen. Der Quotient k/t ist dann ein Maß für die Fläche des Viertelkreises. Sein Vierfaches also ein Näherungswert für π.

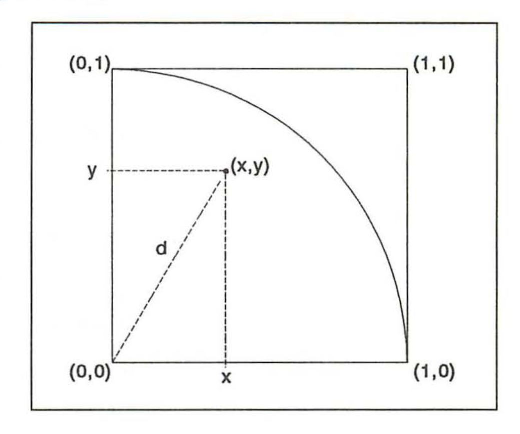


Bild 1.12: »Pi durch Zufallsregen«

Einen Regentropfen mit den Koordinaten (x,y) mit $0 \leq x \leq 1$ und $0 \leq y \leq 1$ erzeugt man mit einem Zufallsgenerator. Hierzu bietet Megamax-Modula die Prozedur `Random` aus dem Modul `RandomGen` an.

```
MODULE PiDurchZufallsRegen;

FROM InOut     IMPORT Read, WriteReal, WriteCard, WriteString,
                      WriteLn, KeyPressed;
FROM RandomGen IMPORT Randomize, Random;

VAR
    Tropfen, KreisTropfen : LONGCARD;
    x, y                  : REAL;
    Taste                 : CHAR;

BEGIN
  Tropfen :=0D;
  Randomize(0D);   (* Zufallsgenerator mit zufälligem Wert initialisieren *)
  WHILE ( Tropfen < MAX(LONGCARD) ) AND NOT KeyPressed() DO
    INC(Tropfen);
    x := Random();
    y := Random();
    IF x*x + y*y < 1.0 THEN INC(KreisTropfen) END
  END;
  WriteString("Insgesamt regnete es "); WriteCard(Tropfen,1);
  WriteString(" Tropfen,"); WriteLn;
  WriteString("davon trafen "); WriteCard(KreisTropfen,1);
  WriteString(" den Viertelkreis."); WriteLn;
```

```
    WriteString("Der Näherungswert für Pi durch Zufallsregen beträgt also ");
    WriteReal(4.0 * FLOAT(KreisTropfen) / FLOAT(Tropfen),15,13);
    Read(Taste); Read(Taste)
END PiDurchZufallsRegen.
```

Ob ein Tropfen in den Viertelkreis gefallen ist, wird mit seinem Abstand

$$d = \sqrt{x^2 + y^2}$$

zum Nullpunkt überprüft. Ist $d < 1$, so liegt der Tropfen im Kreis. Dies ist hier gleichbedeutend mit $(x^2 + y^2) < 1$.

Zum vorläufigem Abschluß der Arbeit mit `REAL`-Zahlen fassen wir zusammen:

1. In einem Ausdruck dürfen nicht `REAL`- mit `INTEGER`- oder `CARDINAL`-Variablen gemischt vorkommen. Man muß vorher konvertieren und benutzt dazu die Transferfunktionen
   `FLOAT` (bei einigen Compilern für Langzahlen `FLOATD`),
   `TRUNC` (bei einigen Compilern für Langzahlen `TRUNCD`)

2. Weitere Standardfunktionen für `REAL` sind:
   `ABS` (Absolutbetrag), `MIN` und `MAX`.
   Achtung: `DEC` und `INC` arbeiten nicht mit `REAL`!
   Im übrigen gibt es noch einige Funktionen im Modul `MathLib0` oder `MathLib`, unter anderem trigonometrische Funktionen.

3. Die Ein- und Ausgabe von `REAL`-Werten erfolgt mit `ReadReal` und `WriteReal` aus dem Modul `InOut`; bei manchen Systemen stehen diese Funktionen allerdings in einem extra Modul `RealInOut`. Bei der Eingabe über diese Funktionen kann der Dezimalpunkt entfallen.

4. `REAL`-Konstanten sind mit Dezimalpunkt zu schreiben: also `3.0` statt `3`. Sehr große und sehr kleine Zahlen schreibt man mit Exponent: z.B. `3.0E11`, `-4.8E-14` (bedeutet $3*10^{11}$ bzw. $-4.8*10^{-14}$).

1.3.5 Der Datentyp BOOLEAN

Dieser Datentyp (benannt nach dem Mathematiker *Georges Boole*) repräsentiert die Wahrheitswerte `FALSE` (falsch) und `TRUE` (wahr). Als Anwendungsgebiet von `BOOLEAN`-Variablen ist das Steuern von Schleifen und bedingten Anweisungen zu nennen. Boolesche Ausdrücke haben Sie bereits in einer Schleife kennengelernt:

```
REPEAT <...> UNTIL x=y;
```

Hier stellt »x=y« einen Booleschen Ausdruck dar. Das Ergebnis eines solchen Vergleichs läßt sich einer Variablen vom Typ BOOLEAN zuweisen! Wie, sehen Sie in diesem Abschnitt.

Rechnerintern wird eine BOOLEAN-Variable in einem Byte (Hänisch-, SPC- und TDI-Modula) oder zwei Byte (MSM2, Megamax-Modula) dargestellt:

FALSE entspricht dem Bitmuster 00000000 und
TRUE entspricht dem Bitmuster 00000001.

Wie sie sehen, wird die entscheidende Information nur in einem einzigem Bit dargestellt. Mehr ist eigentlich nicht nötig!

Folgende Operatoren sind auf BOOLEAN-Variablen anwendbar:

1. Zweistellige Operatoren:
   AND: p AND q oder p & q: p und q sind wahr
   OR: p OR q: p oder q (oder beide) sind wahr
2. Einstellige Operatoren:
   NOT: NOT p oder ~p: nicht p (p ist falsch)
3. Vergleichs-Operatoren (Relationen):
   <, >, <=, >=, =, # (oder <>)

   es gilt: FALSE < TRUE (siehe oben interne Darstellung!)

Ähnlich wie bei Zahlen, bei denen das Vorzeichen am stärksten bindet, danach * und / (bzw. DIV, MOD) Vorrang hat vor + und -, gibt es auch für die obigen Operatoren Vorrangregeln in Ausdrücken: NOT hat den höchsten Rang, es folgt AND (»logische Multiplikation«), dann OR (»logische Addition«). Den geringsten Rang haben die Relationen.

Relationen liefern immer ein Ergebnis vom Typ BOOLEAN. Die Argumente können aber außer vom Typ BOOLEAN noch von den folgenden Typen sein: INTEGER / LONGINT, CARDINAL / LONGCARD, REAL / LONGREAL, CHAR, Aufzählungs- und Unterbereichstypen und teilweise Mengen (BITSET, SET OF...). Die beiden Argumente müssen natürlich von demselben Typ sein:

```
7 < 12          TRUE
7 = 12          FALSE
7.2 # 12.0      TRUE
```

Wegen des geringen Vorranges der Relationen nach AND und OR muß man für »falls a<b oder c# d dann...« in Modula Klammern benutzen:

```
IF (a < b) OR (c # d) THEN <...> END;
```

Wichtig: die Auswertung eines Booleschen Ausdruckes wird abgebrochen, sobald sein Ergebnis fest steht. Im Klartext: `p AND q` ist auf jeden Fall falsch, falls `p` bereits `FALSE` ist. Der Wahrheitswert von `q` wird in diesem Fall nicht mehr geprüft! Daher erzeugt die Abfrage

```
IF (a # 0) AND (b DIV a < 10) THEN <...> END;
```

keinen Fehler »Divison durch 0«, falls `a=0` ist! Vertauscht man die Argumente von `AND`:

```
IF (b DIV a < 10) AND (a # 0) THEN ... END;
```

so tritt für `a=0` sofort der Fehler auf.

`p OR q` ist auf jeden Fall wahr, falls `p` bereits `TRUE` ist, und `q` wird dann nicht mehr geprüft. Dies sollte man beachten, wenn auf eine hohe Ablaufgeschwindigkeit Wert gelegt wird.

Weiter Tips:

1. Der Ausdruck

   ```
   IF p = TRUE THEN <...> END;
   ```

   ist unsinnig, denn es reicht ja

   ```
   IF p THEN <...> END;
   ```

   Folgende Konstruktion findet man selbst in Profi-Programmen:

   ```
   IF p = TRUE THEN q := FALSE
               ELSE q := TRUE
   END;
   ```

   Was natürlich viel übersichtlicher gelöst wird mit:

   ```
   q := NOT p;
   ```

2. Es gelten die `DE MORGAN`'schen Gesetze:

   ```
   (NOT p) AND (NOT q) = NOT (p OR q)
   (NOT p) OR (NOT q)  = NOT (p OR q)
   ```

3. Die logische Antivalenz (»entweder p oder q«, in manchen Sprachen als `p XOR q`) realisiert man in Modula mit `p#q`. Genauso läßt sich die Äquivalenz (»p genau dann wenn q«) mit `p = q` ausdrücken.

4. Weil `FALSE<TRUE` ist, läßt sich die logische Implikation (»Wenn p, dann q« oder »aus p folgt q«) mit `p<=q` oder `NOT p OR q` realisieren.

5. Weil `FALSE<TRUE` ist, können `BOOLEAN`-Variablen in `FOR`-Schleifen verwendet werden:

   ```
   FOR p := FALSE TO TRUE DO <...> END;
   ```

   die Schleife läuft genau zwei Mal.

6. Mit `VAR b: BOOLEAN;` sind folgende Ausdrücke korrekt:

`b:=FALSE;`	`TRUE` und `FALSE` sind vordefinierte Konstanten
`b:=5<7;`	ergibt `TRUE`
`b:=a#b;`	mit a,b vom Typ `REAL`
`b:=(10>2) AND (0<7);`	ergibt `TRUE`.

7. Für den mathematischen Ausdruck `0<i<10` schreibt man in Modula:

   `(0<i) AND (i<10)` (i vom Typ `INTEGER`)

Das folgende Programm bringt eine Wahrheitstabelle der Form:

p	q	p AND q	p OR q	NOT p	p <= q
falsch	falsch	falsch	falsch	wahr	wahr
falsch	wahr	falsch	wahr	wahr	wahr
wahr	falsch	falsch	wahr	falsch	falsch
wahr	wahr	wahr	wahr	falsch	wahr

```
MODULE WahrheitsTabelle;

FROM InOut IMPORT WriteString, WriteLn, Read;

VAR p, q  : BOOLEAN;
    taste : CHAR;

BEGIN
  WriteString("   p    |    q    |p AND q | p OR q |  NOT p | p <= q |");
  WriteLn;
  WriteString("-------------------------------------------------------");
  WriteLn;
  FOR p:=FALSE TO TRUE DO
```

```
    FOR q:=FALSE TO TRUE DO
      IF p      THEN WriteString("  wahr  |") ELSE WriteString(" falsch |") END;
      IF q      THEN WriteString("  wahr  |") ELSE WriteString(" falsch |") END;
      IF p & q  THEN WriteString("  wahr  |") ELSE WriteString(" falsch |") END;
      IF p OR q THEN WriteString("  wahr  |") ELSE WriteString(" falsch |") END;
      IF NOT p  THEN WriteString("  wahr  |") ELSE WriteString(" falsch |") END;
      IF p <= q THEN WriteString("  wahr  |") ELSE WriteString(" falsch |") END;
      WriteLn
    END
  END;
  Read(taste);
END WahrheitsTabelle.
```

1.3.6 Der Datentyp CHAR

Dieser Typ dient zur Speicherung eines Zeichens (CHAR steht für *character*, »Zeichen«). Er umfaßt die Zeichen des ASCII (*American Standard Code for Information Interchanging*, »Amerikanischer Code für Informationsaustausch«) und enthält noch einige Atari-spezifische zusätzliche Zeichen. Zur Abspeicherung wird ein Byte verwendet, das macht 256 verschiedene Zeichen (die Zeichen sind im Anhang aufgelistet). Da der Typ CHAR durch die Binärdarstellung »durchnumeriert« ist, kann man alle Vergleichs-Operatoren auf CHAR-Ausdrücke anwenden. Aus demselben Grund können CHAR-Variablen auch in FOR-Schleifen eingesetzt werden.

Folgende Standardfunktionen sind anwendbar:

CAP(ch) Wenn ch ein Kleinbuchstabe ist, liefert CAP(ch) den entsprechenden Großbuchstaben. Andere Zeichen behalten ihren Wert.

DEC(ch) erniedrigt ch (voriger ASCII-Wert).

INC(ch) erhöht ch (nächster ASCII-Wert).

CHR(i) wandelt die CARDINAL-Zahl i in das i-te Zeichen des Zeichensatzes um. Beispiel: CHR(65) ergibt "A".

ORD(ch) ist die Umkehrfunktion zu CHR, es wird der ASCII-Wert des Zeichens zurückgegeben. Es gilt also (für i<256):
CHR(ORD(ch))=ch und
ORD(CHR(i))=i.

Des weiteren liefert MIN(CHAR) und MAX(CHAR) das kleinste bzw. größte Zeichen (CHR(0) und CHR(255)).

In Modula kann ein Zeichen in drei Möglichkeiten dargestellt werden. Wir demonstrieren es hier am Großbuchstaben "A":

1: `WRITE("A");`	nur für druckbare ASCII-Zeichen
2: `WRITE(CHR(65));`	jedes beliebige Zeichen
3: `WRITE(101C);`	mit oktaler Zeichen-Konstante

Beim letzten Punkt werden die Zeichen oktal durchnumeriert und ein »C« (für CHAR) angehängt. Die oktale Zahl »101« entspricht der Dezimalzahl 65 ($1*8^2 + 0*8 + 1*1 = 65$), also stellt 101C das 65. Zeichen dar, nämlich »A«.

Ein kleiner Trick: Da die Ziffern 0,1,2...9 die aufeinanderfolgenden ASCII-Werte 48,49,...57 haben, ist es möglich, ein Zahlzeichen in die entsprechende Ziffer umzuwandeln, also beispielsweise aus dem Zeichen "3" die CARDINAL-Zahl 3 zu machen:

```
i := ORD(ch) - ORD("0");
```

Das Einlesen einer CARDINAL-Variablen i funktioniert im Prinzip so:

```
i := 0;
Read(ch);
WHILE ("0" <= ch) AND (ch <= "9") DO
  i := 10 * i + ORD(ch) - ORD("0");
  Read(ch)
END;
```

Das folgende Programm druckt die ASCII-Zeichen mit den Nummern 32 bis 255 aus; das sind die druckbaren Zeichen. Sie werden mit ihrem dezimalen und oktalen Werten angegeben.

```
MODULE AsciiTabelle;

FROM InOut IMPORT Write, WriteString, WriteCard, WriteNum, WriteLn, Read;

VAR ch, taste : CHAR;

BEGIN
   FOR ch := " " TO CHR(255) DO                  (* nur druckbare Zeichen *)
      WriteString("  "); Write(ch); WriteString("  ");
      WriteCard(ORD(ch),3);  WriteString("  ");          (* ascii dezimal *)
      WriteNum(ORD(ch),8,3); WriteString("C   |");               (* oktal *)
      (* für andere Compiler: 'WriteOct(ORD(ch),3)' statt 'WriteNum...'   *)
      IF (ORD(ch) MOD 64 = 63) THEN Read(taste) END;
      IF (ORD(ch) MOD  4 =  3) THEN WriteLn END;
   END;
END AsciiTabelle.
```

	32	40C	!	33	41C	"	34	42C	#	35	43C
$	36	44C	%	37	45C	&	38	46C	'	39	47C
(	40	50C	)	41	51C	*	42	52C	+	43	53C
,	44	54C	-	45	55C	.	46	56C	/	47	57C
0	48	60C	1	49	61C	2	50	62C	3	51	63C
4	52	64C	5	53	65C	6	54	66C	7	55	67C
8	56	70C	9	57	71C	:	58	72C	;	59	73C
<	60	74C	=	61	75C	>	62	76C	?	63	77C
@	64	100C	A	65	101C	B	66	102C	C	67	103C
D	68	104C	E	69	105C	F	70	106C	G	71	107C
H	72	110C	I	73	111C	J	74	112C	K	75	113C
L	76	114C	M	77	115C	N	78	116C	O	79	117C
P	80	120C	Q	81	121C	R	82	122C	S	83	123C
T	84	124C	U	85	125C	V	86	126C	W	87	127C
X	88	130C	Y	89	131C	Z	90	132C	[	91	133C
\	92	134C	]	93	135C	^	94	136C	_	95	137C
`	96	140C	a	97	141C	b	98	142C	c	99	143C
d	100	144C	e	101	145C	f	102	146C	g	103	147C
h	104	150C	i	105	151C	j	106	152C	k	107	153C
l	108	154C	m	109	155C	n	110	156C	o	111	157C
p	112	160C	q	113	161C	r	114	162C	s	115	163C
t	116	164C	u	117	165C	v	118	166C	w	119	167C
x	120	170C	y	121	171C	z	122	172C	{	123	173C
		124	174C	}	125	175C	~	126	176C	Δ	127	177C
Ç	128	200C	ü	129	201C	é	130	202C	â	131	203C
ä	132	204C	à	133	205C	å	134	206C	ç	135	207C
ê	136	210C	ë	137	211C	è	138	212C	ï	139	213C
î	140	214C	ì	141	215C	Ä	142	216C	Å	143	217C
É	144	220C	æ	145	221C	Æ	146	222C	ô	147	223C
ö	148	224C	ò	149	225C	û	150	226C	ù	151	227C
ÿ	152	230C	Ö	153	231C	Ü	154	232C	¢	155	233C
£	156	234C	¥	157	235C	ß	158	236C	ƒ	159	237C
á	160	240C	í	161	241C	ó	162	242C	ú	163	243C
ñ	164	244C	Ñ	165	245C	ª	166	246C	º	167	247C
¿	168	250C	⌐	169	251C	¬	170	252C	½	171	253C
¼	172	254C	¡	173	255C	«	174	256C	»	175	257C
ã	176	260C	õ	177	261C	Ø	178	262C	ø	179	263C
œ	180	264C	Œ	181	265C	À	182	266C	Ã	183	267C
Õ	184	270C	¨	185	271C	´	186	272C	†	187	273C
¶	188	274C	©	189	275C	®	190	276C	™	191	277C
ĳ	192	300C	Ĳ	193	301C	א	194	302C	ב	195	303C
ג	196	304C	ד	197	305C	ה	198	306C	ו	199	307C
ז	200	310C	ח	201	311C	ט	202	312C	י	203	313C
כ	204	314C	ל	205	315C	מ	206	316C	נ	207	317C
ס	208	320C	ע	209	321C	פ	210	322C	צ	211	323C
ק	212	324C	ר	213	325C	ש	214	326C	ת	215	327C
ן	216	330C	ך	217	331C	ם	218	332C	ף	219	333C
ץ	220	334C	§	221	335C	∧	222	336C	∞	223	337C
α	224	340C	β	225	341C	Γ	226	342C	π	227	343C
Σ	228	344C	σ	229	345C	µ	230	346C	τ	231	347C
Φ	232	350C	Θ	233	351C	Ω	234	352C	δ	235	353C
∮	236	354C	ϕ	237	355C	∈	238	356C	∩	239	357C
≡	240	360C	±	241	361C	≥	242	362C	≤	243	363C
⌠	244	364C	⌡	245	365C	÷	246	366C	≈	247	367C
°	248	370C	•	249	371C	·	250	372C	√	251	373C
ⁿ	252	374C	²	253	375C	³	254	376C	¯	255	377C

Erweiterte ASCII-Tabelle

Die Zeichen ab Nummer 128 (also ab `CHR(128)`) gehören nicht mehr zur ASCII-Norm. Sie stimmen beim Atari weitgehend mit dem IBM-Zeichensatz überein, so daß IBM-kompatible Drucker die Mehrzahl dieser Zeichen richtig drucken können. Dies stimmt für die Umlaute, nicht aber für »ß«.

Die Zeichen `CHR(0)` bis `CHR(31)` (also die ersten 32) sind für bestimmte Steuerzwecke reserviert. Sie lösen bei dem Gerät (Drucker, Bildschirm...) auf das sie ausgegeben werden, gewisse Funktionen aus. Bei einem Drucker kann man damit zum Beispiel den linken Rand einstellen oder den Zeichensatz wechseln (siehe Beispielprogramm »Druck« im Kapitel 4.4).

Der Atari-Bildschirm emuliert die »Escape-Sequenzen« des VT52-Terminals. Eine solche Sequenz wird durch das »ESC«-Zeichen (`33C`) und einem weiteren Zeichen ausgelöst und bewirkt eine Aktion auf dem Bildschirm. Die Zeichen können mit zwei `Write`-Aufrufen ausgegeben werden. Beispielsweise löscht die Sequenz ESC-"E" den Bildschirm. In Modula programmiert man dazu:

```
Write(33C); Write("E"); (*Write aus Terminal*)
```

Wir haben die Escape-Sequenz ESC-"Y" bereits beim Schreiben der Prozedur `GotoXY` im Abschnitt 1.3.3 benutzt.

Weitere interessante Escape-Sequenzen:

ESC-"A" bewegt den Cursor eine Zeile nach oben
ESC-"B" bewegt den Cursor eine Zeile nach unten
ESC-"C" bewegt den Cursor ein Zeichen nach rechts
ESC-"D" bewegt den Cursor ein Zeichen nach links
ESC-"K" löscht die Zeile ab der Cursorposition
ESC-"f" der Cursor wird unsichtbar
ESC-"e" der Cursor wird sichtbar
ESC-"p" bewirkt weiße Schrift auf schwarzem Grund
ESC-"q" hebt ESC-"p" auf

1.3.7 Der Datentyp BITSET

Dieser Datentyp repräsentiert die Menge `{0,1,2,...,15}` (engl. *set* = dt. »Menge«, also `BITSET`: »Menge von Bits«). Er wird intern in 2 Byte abgespeichert. Jedes Bit steht für ein Element. Eine 1 bedeutet, das Element ist vorhanden; eine 0, es ist nicht in der Menge. Die Menge `{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}`, die alle Elemente ihres Types enhält, wird also mit `11111111 11111111` dargestellt, die leere Menge `{}` mit dem Bitmuster `00000000 00000000`. Als Operatoren sind definiert:

\+ Mengenvereinigung: {2,3}+{3,4} ergibt {2,3,4}.

\- Mengendifferenz: {2,3}-{3,4} ergibt {2}.

* Mengendurchschnitt: {2,3}*{3,4} ergibt {3}.

/ Symmetrische Differenz: {2,3}/{3,4} ergibt {2,4}.

Der Operator IN bedeutet »ist enthalten in«, erwartet als Argumente ein CARDINAL-Element und eine BITSET-Menge und liefert als Ergebnis einen Wert vom Typ BOOLEAN. Alles klar? Eigentlich ganz einfach: der (Boolesche) Ausdruck

a IN B entspricht mathematisch $a \in B$.

Beispiel: 3 IN {1,2,3} ergibt TRUE und
3 IN {2,4,6} ergibt FALSE.

So kann man mit Mengen auch ein paar Formeln aufstellen:

i IN (m + n)	bedeutet	(i IN m) OR (i IN n)
i IN (m - n)	bedeutet	(i IN m) AND NOT (i IN n)
	oder so:	(i IN m) > (i IN n) (nachdenken!)
i IN (m * n)	bedeutet	(i IN m) AND (i IN n)
i IN (m / n)	bedeutet	(i IN m) # (i IN n)

Außerdem sind auf Mengen noch die Standardprozeduren INCL (Inklusion) und EXCL (Exklusion) anwendbar:

nach m={2,3}; INCL(m,4) ist m gleich {2,3,4},
nach m={2,3}; EXCL(m,3) ist m gleich {2}.

Was kann man nun mit einem solchen Datentyp anfangen?

Das Interessante an dem Typ BITSET ist, daß man auf einzelne Bits eines gespeicherten »Wortes« zugreifen kann. Ein »Wort« (engl. *word)* ist dabei eine Einheit aus zwei Bytes. Der Datentyp WORD ist in Modula definiert. Er ist zwar nicht Bestandteil der Sprache selbst, sondern wird im Modul SYSTEM – auf den wir noch zu sprechen kommen – deklariert. Der WORD-Typ ist sehr flexibel: Man kann ihn jedem beliebigen 2-Byte-Typ zuweisen, also zum Beispiel CARDINAL, INTEGER oder BOOLEAN (bei Megamax). Dies funktioniert aber nur bei der Übergabe in der »Parameterliste einer Prozedur«.

Eine Prozedur bezeichnet ein Unterprogramm. Das ist ein Programmteil, der durch Nennung seines Namens im »Hauptprogramm« abgearbeitet wird (»Prozeduraufruf«). Eine Prozedur kann eine Parameterliste haben. Darin sind die Daten (mit Typbezeichnung) aufgelistet, mit denen die Prozedur arbeitet. Prozeduren werden ausführlich in Abschnitt 1.5 behandelt. Das nachstehende Beispielprogramm verdeutlicht bereits das Prinzip.

Es geht hierbei darum, das bitweise »und«, »oder«, »entweder...oder« und »nicht« zu simulieren, daß sie vielleicht von C- oder Turbo-Pascal für ganze Zahlen her kennen. Schauen Sie sich an, wie elegant man das in Modula machen kann!

```
MODULE BitSetTest;

FROM SYSTEM IMPORT WORD;
FROM InOut  IMPORT Read, ReadCard, WriteCard, WriteLn, WriteString;

PROCEDURE und (a,b : WORD) : WORD;
BEGIN
  RETURN WORD(BITSET(a) * BITSET(b))
END und;

PROCEDURE oder(a,b : WORD) : WORD;
BEGIN
  RETURN WORD(BITSET(a) + BITSET(b))
END oder;

PROCEDURE exOder(a,b : WORD) : WORD;
BEGIN
  RETURN WORD(BITSET(a) / BITSET(b))
END exOder;

PROCEDURE nicht(a:WORD) : WORD;
BEGIN
  RETURN WORD(BITSET(MAX(WORD)) - BITSET(a))
END nicht;

VAR i,j   : CARDINAL;
    Taste : CHAR;

BEGIN
  WriteString("Programm zum Testen von BITSET und der Mengenoperationen");
  WriteLn; WriteLn;
  WriteString("1. Zahl i  : "); ReadCard(i);
  WriteString("2. Zahl j  : "); ReadCard(j); WriteLn;
  WriteString("i und  j   = "); WriteCard(CARDINAL(und (i,j)),5);
  WriteLn;
  WriteString("i oder j   = "); WriteCard(CARDINAL(oder (i,j)),5);
  WriteLn;
  WriteString("i exOder j = "); WriteCard(CARDINAL(exOder(i,j)),5);
  WriteLn;
  WriteString("nicht i    = "); WriteCard(CARDINAL(nicht(i)),5);
```

```
  Read(Taste)
END BitSetTest.
```

Beispiel: Die Eingabe i = 13 und j = 7 erzeugt:

```
i und j      =    5
i oder j     =   15
i exOder j   =   10
nicht i      =   65522
```

Prüfen Sie diese Ergebnisse anhand der Bitmuster nach!

Man sieht an dem Programm:

1. Der Typtransfer zum Typ `BITSET` wird mit `BITSET(var)`, wobei `var` eine beliebige 2-Byte-Variable ist, erledigt.
2. Die gesamte Bitmanipuliererei, die Sie von Lowlevel-Sprachen her kennen, ist bequem in Modula möglich!
3. Für den Datentyp `BITSET` gibt es keine Ein- und Ausgabemöglichkeit.

Wenn Sie sehen wollen, wie eine bestimmte Menge intern abgespeichert wird, hilft folgendes Vorgehen, mit dem man die interne Abspeicherung beliebiger 2-Byte-Variablen (`CARDINAL`, `INTEGER`, `BITSET`) erfassen kann. Wir verwenden wieder den Joker-Datentyp `SYSTEM.WORD`:

```
MODULE BitMuster;

FROM InOut  IMPORT WriteLn, WriteString, Write, Read;
FROM SYSTEM IMPORT WORD;

PROCEDURE SchreibBits( w : WORD);
VAR i : CARDINAL;
BEGIN
  FOR i:=7 TO 0 BY -1 DO                     (* zunächst das untere Byte, *)
    IF i IN BITSET(w) THEN Write("1") ELSE Write("0") END
  END;
  Write(" ");
  FOR i:=15 TO 8 BY -1 DO                    (* ...   dann das obere Byte *)
    IF i IN BITSET(w) THEN Write("1") ELSE Write("0") END
  END;
END SchreibBits;
```

```
VAR c  : CARDINAL;
    i  : INTEGER;
    b  : BOOLEAN;
    bs : BITSET;
    ch : CHAR;

BEGIN
  c  := 5;       WriteString("5       intern: "); SchreibBits(c);  WriteLn;
  i  := -1;      WriteString("-1      intern: "); SchreibBits(i);  WriteLn;
  b  := TRUE;    WriteString("TRUE    intern: "); SchreibBits(b);  WriteLn;
  bs := {1,3,5}; WriteString("{1,3,5} intern: "); SchreibBits(bs); WriteLn;
  Read(ch)
END BitMuster.
```

Das Programm liefert folgende Ausgabe:

```
5         intern: 00000000 00000101
-1        intern: 11111111 11111111
TRUE      intern: 00000000 00000001
{1,3,5}   intern: 00101010 00000000
```

1.3.8 Zweck und Form einer Konstantendeklaration

Wir haben bereits in den bisherigen Beispielprogrammen Konstantendeklarationen benutzt. Hier nur noch einmal eine Zusammenfassung und Vertiefung.

Soll ein Bezeichner für einen festen Wert stehen, so muß er in einer Konstanten-Deklaration eingeführt werden. Eine Konstantendeklaration hat die Form

```
CONST
    bezeichner1 = Wert1;
    bezeichner2 = Wert2;
    ....
```

Die Werte sind dabei Konstanten von einem beliebigen einfachen Datentyp oder vom Typ `ARRAY OF CHAR` (eine Zeichenkette).

Beispiele:

```
CONST
    Pi = 3.1415926536;               (* REAL-Konstante *)
```

```
PiHalbe = Pi / 2;             (* REAL-Konstante, aus bereits
                                 definierten Konstanten können
                                 also neue Konstanten gebildet
                                 werden *)
MinusPi = - Pi;
Zahl1 = 5;                    (* CARDINAL- oder INTEGER-Konstante *)
Zahl2 = - 5;                  (* INTEGER-Konstante *)
Zahl3 = 5D;                   (* LONGCARD- oder LONGINT- Konstante *)
Zahl4 = 5L;                   (* wie oben bei Megamax-Modula *)
Zahl5 = - 5D;                 (* LONGINT-Konstante *)
Zahl6 = 5.0;                  (* REAL-Konstante (Dezimalpunkt!) *)
Zahl7 = 5.0E6;                (* REAL-Konstante: 5000000.0 *)
Zahl8 = 5.1E-3;               (* REAL-Konstante: 0.0051 *)
ok    = TRUE;                 (* BOOLEAN-Konstante *)
falsch = NOT ok;              (* BOOLEAN-Konstante *)
ESC   = 33C;                  (* CHAR-Konstante (Escape = CHR(27)) *)
Glocke = 7C;                  (* CHAR-Konstante *)
Menge = {1,2,3};              (* BITSET-Konstante *)
Version = "Version 1.0"       (* Zeichenkette *)
Strich = "-----------"        (* wie oben *)
Zahl9 = 14B;                  (* CARDINAL- oder INTEGER-Konstante,
                                 oktale Angabe (= dezimal 12)   *)
Zahl10 = 14H;                 (* CARDINAL oder INTEGER: hexadezimale
                                 Angabe (1*16 + 4*1 = 20)   *)
Zahl11 = 0A1H;                (* Zahlen müssen mit Ziffer beginnen!
                                 (10*16 + 1*1 = 161 dezimal *)
Zahl12 = INTEGER(3)           (* INTEGER, keine CARDINAL-Konstante *)
Zahl13 = CARDINAL(5)          (* nur CARDINAL-Konstante *)
Zahl14 = Zahl1 MOD Zahl12;
Zeiger = NIL;                 (* POINTER-Konstante *)
Leer   = MengenTyp{}          (* Menge, nicht vom Typ BITSET *)
```

Wie man sieht, können bei der Konstanten-Deklaration auch Ausdrücke vorkommen. Bei `Zahl1=5` ist nicht klar, ob 5 als `CARDINAL`- oder `INTEGER`-Konstante gemeint ist. Ein vorangestelltes `CARDINAL` oder `INTEGER` legt aber den Typ fest.

Folgendes ist nicht erlaubt:

`Zeichen=CHR(65);`	Funktionen dürfen nicht benutzt werden
`Zahl1=-e;`	e muß vorher als Konstante definiert sein!
`Zahl2:=7L;`	»:=« ist falsch, es muß »=« heißen
`Zahl3=A1H;`	bei hexadezimal-Konstanten muß das erste Zeichen eine Ziffer sein (hier einfach eine 0 voranstellen).

Auf den Vorteil der Verwendung von Konstantendeklarationen wurde schon hingewiesen:

1. Die Lesbarkeit des Programms erhöht sich.
2. Der Programmtext läßt sich leichter abändern. Wenn zum Beispiel mehrfach im Programmtext `WriteString("Version 1.0")` steht, so müßten bei einem Update alle Stellen gesucht werden und der String ausgetauscht werden. Einfacher wäre es, die Konstante `Version="Version 1.0"` im Deklarationsteil zu ändern.

1.4 Kontrollstrukturen

Die bisherigen Beispielprogramme zeigen bereits des öfteren die Verwendung von Wiederholungsanweisungen (Schleifen) und bedingten Anweisungen. Man faßt die beiden Anweisungstypen unter dem Oberbegriff »Kontrollstrukturen« zusammen, da sie erlauben, vom »linearen Programmablauf« (von »oben« nach »unten«) abzuweichen. Jetzt sollen die Kenntnisse über Kontrollstrukturen vertieft werden.

1.4.1 Wiederholungsanweisungen

Modula kennt vier verschiedene Schleifentypen, die mit den Schlüsselworten `REPEAT`, `WHILE`, `LOOP` und `FOR` eingeleitet werden.

Die REPEAT-Schleife

Diese Schleife hat die Form

```
REPEAT
  <Anweisungsfolge>
UNTIL <Boolescher Ausdruck>;
```

Zu deutsch etwa: Wiederhole *<Anweisungsfolge>* bis *<Bedingung erfüllt>*.

Da die Abbruchsbedingung am Ende der Schleife getestet wird, wird eine `REPEAT`-Schleife mindestens einmal durchlaufen. Die Schleife

```
REPEAT
  <Anweisungsfolge>
UNTIL FALSE;
```

ist eine »Endlosschleife.« Da `FALSE` natürlich immer »falsch« ist, wird die Abbruchsbedingung nie erfüllt und die Schleife hört nicht mehr auf.

In der Anweisungsfolge muß auf die Endebedingung eingewirkt werden, damit die Abbruchsbedingung einmal `TRUE` wird, sonst läuft die Schleife »ewig«. Die folgende Schleife druckt die ungeraden Zahlen von 1 bis 99:

```
VAR i: CARDINAL;
<...>
i := 1;                  (* i fängt bei 1 an *)
REPEAT
  WriteCard(i,3);        (* i ausdrucken *)
  INC(i,2)               (* i um 2 erhöhen *)
UNTIL i = 101;           (* letzte gedruckte Zahl ist 99 *)
```

Nie erreicht würde hingegen die Abbruchsbedingung

```
UNTIL i = 100;
```

da `i` in der Schleife nur ungerade Werte annimmt. Man hätte hier – ungewollt – eine Endlosschleife.

Die Schleife

```
REPEAT
  (* nix *)
UNTIL KeyPressed();
```

läuft so lange, bis eine Taste gedrückt ist, sie wartet also auf einen Tastendruck. Besser ist die Lösung

```
VAR taste : CHAR;
....
READ(taste);
```

da der Rechner während des Wartens noch andere Dinge (Drucker-Spooler) erledigen kann.

Die WHILE-Schleife

Dieser Schleifentyp ist vergleichbar mit der `REPEAT`-Schleife. Sie hat die Form

```
WHILE <Boolescher Ausdruck> DO
      <Anweisungsfolge>
END;
```

Zu deutsch etwa: Solange <*Bedingung*> erfüllt ist, führe <*Anweisungen*> aus.

Im Gegensatz zur `REPEAT`-Schleife wird also das Abbruchskriterium am Anfang der Schleife geprüft. Dadurch kann es sein, daß die Schleife keinmal ausgeführt wird. Bei der Abbruchsbe-

dingung gibt es noch einen kleinen Unterschied: Die REPEAT-Schleife bricht ab, wenn die Abbruchsbedingung TRUE wird, die WHILE-Schleife, wenn die Abbruchsbedingung FALSE wird (da sie läuft, solange die Bedingung erfüllt ist). Die Schleife

```
WHILE TRUE DO <Anweisungen> END;
```

ist also eine Endlosschleife.

Die REPEAT-Schleife und die WHILE-Schleife kennen Sie schon von vorhergehenden Programmen; daher erfolgt hier kein weiteres Beispiel.

Die LOOP-Schleife

Das englische Word *loop* heißt direkt übersetzt »Schleife«. Dieser Schleifentyp ist bis jetzt noch nicht vorgekommen. Es handelt sich um eine recht allgemeine Schleife. Die Abbruchsbedingung kann nämlich irgendwo innerhalb des Anweisungsblocks stehen. Die Anweisung EXIT (engl. *exit* = »Ausgang«) beendet die Schleife. In einer LOOP-Schleife sind auch mehrere EXIT-Anweisungen möglich; die Schleife kann also mehrere Abbruchsbedingungen an verschiedenen Stellen haben:

```
LOOP
  <Anweisungsfolge1>
  IF <Bedingung1> THEN EXIT END;
  <Anweisungsfolge2>
  IF <Bedingung2> THEN EXIT END;
  <Anweisungsfolge3>
END;
```

Wenn eine LOOP-Schleife keine EXIT-Anweisung enthält, hat man eine Endlosschleife. Mit der LOOP-Schleife lassen sich alle anderen Schleifen ausdrücken:

Die REPEAT-Schleife

```
REPEAT
  <Anweisungsfolge>
UNTIL <Bedingung>;
```

ist äquivalent zur folgenden LOOP-Schleife:

```
LOOP
  <Anweisungsfolge>
  IF <Bedingung> THEN EXIT END;
END;
```

Ebenso läßt sich die `WHILE`-Schleife:

```
WHILE<Bedingung> DO
  <Anweisungsfolge>
END;
```

umformen zur `LOOP`-Schleife:

```
LOOP
  IF NOT <Bedingung> THEN EXIT END;
  <Anweisungsfolge>
END;
```

Die `LOOP`-Schleife ist etwas primitiver als eine `REPEAT`- bzw. `WHILE`-Schleife. *N. Wirth* schreibt in seinem Buch [W1]: »Obwohl die `LOOP`-Anweisung in einigen Fällen bequem ist, sollte man im Normalfall doch besser eine `WHILE`- bzw. `REPEAT`-Anweisung verwenden, da diese deutlicher eine einzige Endbedingung an einer syntaktisch offensichtlichen Stelle aufzeigen.« Wir folgen diesem Rat.

Fragt sich, wozu die `LOOP`-Schleife überhaupt sinnvoll ist.

Schauen wir uns dazu die Eingabeschleife aus dem Programm »Mittelwertberechnung« aus Abschnitt 1.3.4 an. Sie lautet in Kurzform

```
REPEAT
  <Reelle Zahl einlesen>
  IF zahl # 0.0 THEN
    <Zahl verarbeiten>
  END
UNTIL zahl = 0.0;
```

Dieselbe Zahl wird hier zweimal auf `0.0` geprüft. Das läßt sich vereinfachen:

```
LOOP
  <Reelle Zahl einlesen>
  IF zahl = 0.0 THEN EXIT END;
  <Zahl verarbeiten>
END;
```

Die FOR-Schleife

Diese Wiederholungsanweisung ist sehr komfortabel, weil die Steuerung der Schleife (Inititalisierung, Schrittweite und Abbruchskriterium) bereits im Schleifenkopf vorgegeben sind. Sie hat die Form

```
FOR <Zähler> := <Anfangswert> TO <Endwert> BY <Schrittweite> DO
  <Anweisungsfolge>
END
```

»BY *<Schrittweite>*« kann entfallen.

Wirkung:

1. Die Laufvariable *<Zähler>* wird zu Beginn auf *<Anfangswert>* gesetzt.
2. Es wird geprüft, ob der *<Endwert>* bereits überschritten ist, in dem Fall wird die Schleife beendet.
3. Ansonsten wird *<Anweisungsfolge>* ausgeführt.
4. Anschließend wird die Laufvariable *<Zähler>* um *<Schrittweite>* erhöht.
5. Es wird bei (2.) fortgefahren.

Sind Anfangswert und Endwert gleich, so wird die Anweisungsfolge genau einmal abgearbeitet. Ist bei positiver Schrittweite der Anfangswert höher als der Endwert, so wird die Schleife gar nicht durchlaufen.

Die Schleife

```
FOR ch := "A" TO "Z" BY 2 DO
  WRITE(ch)
END;
```

entspricht

```
ch := "A";
WHILE ch <= "Z" DO
  WRITE(ch);
  INC(ch,2)
END;
```

Es wird ACEGIKMOQSUWY ausgegeben.

Bei einer FOR-Schleife kann die Zählvariable von einem beliebigen skalaren Typ (also CARDINAL, LONGCARD, INTEGER, LONGINT, CHAR, BOOLEAN und Aufzählungstyp (vgl. Abschnitt 1.6.1)) sein. REAL ist nicht zulässig. Wie Sie aus der internen Darstellung von REAL-Variablen wissen, hat eine REAL-Zahl keinen direkten Nachfolger; daher ist auch INC für REAL nicht definiert (DEC ebenso nicht).

Die Schrittweite muß ein Ausdruck von Typ `INTEGER` oder `CARDINAL` sein. Sie kann auch ganz entfallen, als Schrittweite wird dann `1` angenommen.

```
FOR <zähler> := <Anfangswert> TO <Endwert> DO
  <Anweisungsfolge>
END
```

Mit einer negativen Schrittweite (im einfachsten Fall `-1`) kann eine absteigende Schleife organisiert werden (vgl. Prozedur `SchreibeBits` aus Abschnitt 1.3.7).

Es ist zu beachten:

Als Kontrollwerte für Anfangswert, Endwert und Schrittweite sind Ausdrücke zulässig. Folgende Einschränkungen gelten:

- Die Kontrollwerte sollten innerhalb der Schleife nicht verändert werden.
- Die Laufvariable *<zähler>* wird von der `FOR`-Schleife kontrolliert. Sie darf innerhalb der Schleife ebenfalls nicht verändert werden; auf sie darf also nur »lesend« zugegriffen werden.
- Nach Abarbeitung der `FOR`-Schleife ist der Wert der Laufvariablen als undefiniert anzusehen.

Folgendes ist also schlechter Stil und funktioniert bei verschiedenen Compilern unterschiedlich:

```
MODULE SoNicht;

FROM InOut IMPORT WriteLn, Read, WriteCard;

VAR i     : CARDINAL;
    taste : CHAR;

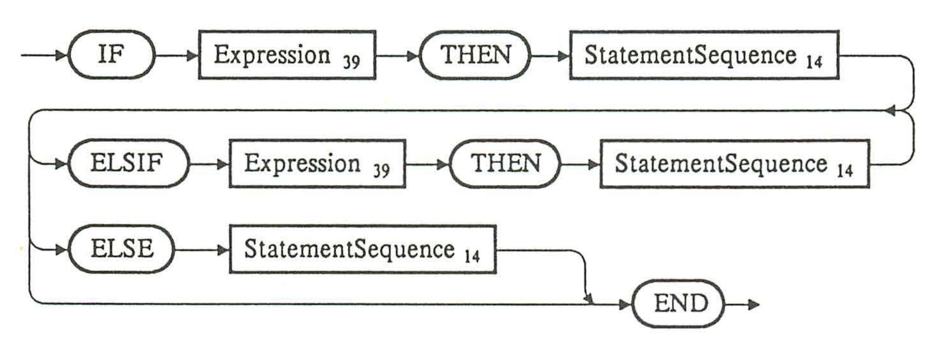
BEGIN
  FOR i := 1 TO 100 BY 3 DO
    WriteCard(i,10);                  (* 'lesender' Zugriff ist ok! *)
    IF i = 10 THEN i := 90 END;               (* das tut man nicht! *)
  END;
  WriteLn;
  WriteCard(i,10);                (* auch das läßt man lieber sein! *)
  Read(taste)
END SoNicht.
```

Wenn man Laufvariable nicht gleichmäßig erhöhen will, sollte eine Konstruktion mit anderen Schleifentypen vorgezogen werden.

1.4.2 Bedingte Anweisungen

Die IF-Anweisung

Diese Anweisung wurde schon oft gebraucht. Ihre Varianten lassen sich am besten aus dem Syntax-Diagramm entnehmen:



SYNTAX: "IfStatement"(47)

Beispiele:

Der einfachste Fall ergibt sich:

```
IF <Bedingung> THEN <Anweisungen> END;
```

Oder mit ELSE-Zweig:

```
IF <Bedingung>
  THEN <Anweisungen1>
  ELSE <Anweisungen2>
END;
```

Das Pseudo-Programmstück

```
IF <b1> THEN <Anweisungen1>
  ELSIF <b2> THEN <Anweisungen2>
  ELSIF <b3> THEN <Anweisungen3>
  ELSE <Anweisungen4>
END;
```

bewirkt, daß Booleschen Ausdrücke `<b1>`, `<b2>`, `<b3>` nacheinander geprüft werden, bis einer von ihnen `TRUE` ergibt. Dann wird die entsprechende Anweisungsfolge ausgeführt und nach dem `END` fortgefahren. Sollten alle Booleschen Ausdrücke `FALSE` ergeben, so wird der `ELSE`-Zweig, also `<Anweisungen4>` ausgeführt.

Das folgende (etwas fiktive) Beispiel demonstriert die Anwendung der IF-Anweisung:

Ein milder Lehrer macht seine Noten auf folgende Weise: Er würfelt mit 3 Würfeln und nimmt die niedrigste der aufgetretenen Augenzahlen als Note. Wieviel Prozent Einser, Zweier, ... Sechser erteilt er? Wir simulieren dies mit Zufallszahlen:

```
MODULE milderLehrer;

FROM RandomGen IMPORT Randomize, RandomCard;
FROM InOut     IMPORT WriteCard, WriteReal, WriteLn, WriteString, KeyPressed;

VAR
    wuerfel1, wuerfel2, wuerfel3, anzahl,
    note1, note2, note3, note4, note5, note6 : CARDINAL;

PROCEDURE SchreibProzent(note : CARDINAL); (* kleiner Trick für Schreibfaule *)
BEGIN
  WriteReal(FLOAT(note)*100.0 / FLOAT(anzahl), 6, 2);
  WriteString("%");
  WriteLn
END SchreibProzent;

BEGIN
  WriteString("Der milde Lehrer würfelt und würfelt, ");
  WriteString("bis Sie eine Taste drücken...");
  anzahl:=0; note1:=0; note2:=0; note3:=0; note4:=0; note5:=0; note6:=0;
  Randomize(12345);
  REPEAT
    wuerfel1:=RandomCard(1,6);
    wuerfel2:=RandomCard(1,6);
    wuerfel3:=RandomCard(1,6);
    IF (wuerfel1=1) OR (wuerfel2=1) OR (wuerfel3=1) THEN INC(note1)
      ELSIF (wuerfel1=2) OR (wuerfel2=2) OR (wuerfel3=2) THEN INC(note2)
      ELSIF (wuerfel1=3) OR (wuerfel2=3) OR (wuerfel3=3) THEN INC(note3)
      ELSIF (wuerfel1=4) OR (wuerfel2=4) OR (wuerfel3=4) THEN INC(note4)
      ELSIF (wuerfel1=5) OR (wuerfel2=5) OR (wuerfel3=5) THEN INC(note5)
      ELSE INC(note6)
    END;
    INC(anzahl);
```

```
    UNTIL KeyPressed();
    WriteLn; WriteLn;
    WriteString("Anzahl der erwürfelten Noten insgesamt:  ");
    WriteCard(anzahl,10);
    WriteLn; WriteLn;
    WriteString("Einsen  : "); SchreibProzent(note1);
    WriteString("Zweien  : "); SchreibProzent(note2);
    WriteString("Dreien  : "); SchreibProzent(note3);
    WriteString("Vieren  : "); SchreibProzent(note4);
    WriteString("Fünfen  : "); SchreibProzent(note5);
    WriteString("Sechsen : "); SchreibProzent(note6);
    REPEAT UNTIL KeyPressed()
  END milderLehrer.
```

Ein Auswertungsbeispiel:

```
Anzahl der erwürfelten Noten insgesamt:         24889
Einsen  : 41.88%
Zweien  : 27.77%
Dreien  : 17.52%
Vieren  :  8.98%
Fünfen  :  3.36%
Sechsen :  0.46%
```

Dies stimmt gut mit den theoretischen Wahrscheinlichkeiten überein.

Die CASE-Anweisung

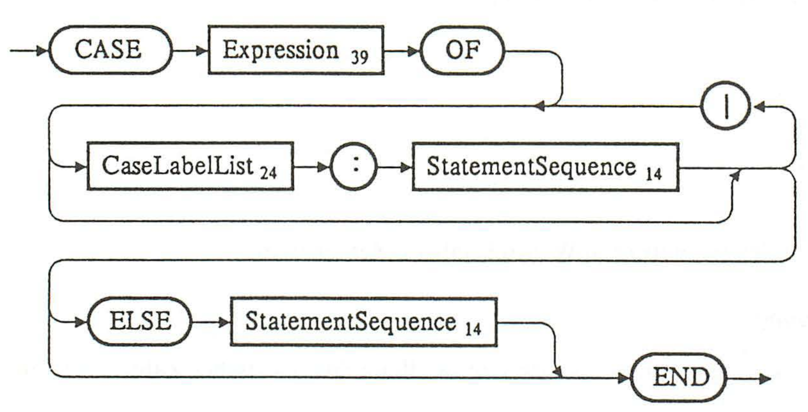
Man betrachte folgendes Programmstück zur Umwandlung eines römischen Zahlzeichens in eine Dezimalzahl:

```
IF Zeichen = "M" THEN Zahl := 1000
ELSIF Zeichen = "D" THEN Zahl := 500
ELSIF Zeichen = "C" THEN Zahl := 100
ELSIF Zeichen = "L" THEN Zahl := 50
ELSIF Zeichen = "X" THEN Zahl := 10
ELSIF Zeichen = "V" THEN Zahl := 5
ELSIF Zeichen = "I" THEN Zahl := 1
ELSE Zahl := 0;
```

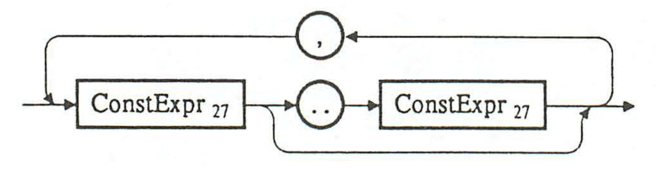
Das sieht sehr umständlich aus und ist schreibintensiv. Modula-2 bietet hier die Fallunterscheidung mittels `CASE`:

```
CASE Zeichen OF
  "M": Zahl := 1000 |
  "D": Zahl := 500 |
  "C": Zahl := 100 |
  "L": Zahl := 50 |
  "X": Zahl := 10 |
  "V": Zahl := 5 |
  "I": Zahl := 1
  ELSE Zahl := 0
END;
```

Mit der CASE-Anweisung können also Verzweigungen von mehreren Fällen, die nur von verschiedenen Werten eines Ausdruckes abhängen, übersichtlich programmiert werden. Die CASE-Anweisung sieht allgemein so aus:



SYNTAX: "CaseStatement"(48)



SYNTAX: "CaseLabelList"(24)

Folgendes ist zu beachten:

- Der Ausdruck nach dem `CASE` darf nur von den skalaren Typen (`CARDINAL`, `INTEGER` (auch `LONGCARD`, `LONGINT`), `CHAR`, `BOOLEAN` sowie deren Aufzählungs- und Unterbereichstypen) sein. `REAL`-Typen sind (wieder einmal) nicht erlaubt!
- Die Marken *`ConstExpr`* dürfen nur Konstanten von diesem Typ sein. Es sind jedoch mehrere Konstanten, die durch », « (Kommata) getrennt sind, erlaubt. Es können auch Bereiche als Marken angegeben werden: `4..7` steht für die Marken `4, 5, 6, 7`.
- Marken dürfen innerhalb einer `CASE`-Anweisung nicht doppelt vorkommen!
- Der `ELSE`-Zweig kann entfallen, wenn sichergestellt ist, daß durch den <*Ausdruck*> immer eine der Marken selektiert wird. Alle Anweisungsfolgen (auch die des `ELSE`-Zweiges) können leer sein, wenn an dieser Stelle nichts ausgeführt werden soll.

Beispiel:

```
CASE i OF
  1    : <Anweisungen1> |
  2,3  : <Anweisungen2> |        (* Mehrere Marken sind erlaubt *)
  4..7 : <Anweisungen3>          (* Steht für 4,5,6,7 *)
  ELSE <Anweisungen>
END;
```

Die CASE-Anweisung kann also nicht in jedem Fall eine geschachtelte IF-Anweisung ersetzten. Im Programm »milder Lehrer« des letzten Abschnittes ist die Fallunterschiedung nur durch Boolesche Ausdrücke und nicht durch Konstanten formulierbar.

Das folgende Beispielprogramm wandelt römische Zahlen in arabische um. Die römischen Zahlen werden dabei als *string* (Zeichenkette) mittels `ReadString` eingegeben. Wir benutzen dazu den Datentyp `String` aus dem Modul `Strings`. Aus diesem Modul stammt auch die Funktion `Length`, die die Länge eines Strings angibt. Ein String ist also eine Folge von Zeichen. Auf das `i`-te Zeichen eines Strings `s` greift man mit `s[i]` zu.

```
MODULE RoemischeZahlenKonvertierung;

FROM InOut   IMPORT ReadString, WriteString, WriteCard, WriteLn;
FROM Strings IMPORT String, Length;

VAR RoemZahlWort                         : String;
    AktuellZiffer, LetzteZiffer, zahl : CARDINAL;
    sub                                  : BOOLEAN;      (* subtrahieren *)
    i                                    : INTEGER;
```

```
BEGIN
  REPEAT
    WriteLn;
    WriteString("RÖMISCHE ZAHL ( 'RETURN' = ENDE ): ");
    ReadString(RoemZahlWort);
    WriteLn;
    zahl:=0;
    LetzteZiffer:=0;
    FOR i := INTEGER(Length(RoemZahlWort)) - 1 TO 0 BY -1 DO
      CASE CAP(RoemZahlWort[i]) OF
        'M' : AktuellZiffer :=1000|
        'D' : AktuellZiffer :=500|
        'C' : AktuellZiffer :=100|
        'L' : AktuellZiffer :=50|
        'X' : AktuellZiffer :=10|
        'V' : AktuellZiffer :=5|
        'I' : AktuellZiffer :=1
      END;
      sub:=(LetzteZiffer>AktuellZiffer);
      IF sub THEN zahl:=zahl-AktuellZiffer ELSE zahl:=zahl+AktuellZiffer END;
      LetzteZiffer:=AktuellZiffer;
    END;
    WriteString("In arabischerSchreibweise: "); WriteCard(zahl,1);
  UNTIL zahl = 0;
END RoemischeZahlenKonvertierung.
```

Die Eingabe `MCMLXXXIX` liefert `1989`.

1.5 Das Prozeduren-Konzept

1.5.1 Parameterlose Prozeduren

Bei größeren Programmen ist es sinnvoll, das zu bearbeitende Problem in Teilaufgaben zu gliedern. Diese Teilaufgaben lassen sich in Modula als »Prozeduren« (Unterprogramme) formulieren.

Eine Prozedur besteht aus

1. dem Prozedur-Kopf: Er sieht bei paramterlosen Prozeduren so aus:
   `PROCEDURE <ProzedurName>;`
   wobei *<ProzedurName>* ein beliebiger (noch nicht benutzter und nicht reservierter) Bezeichner ist.

2. einem Deklarationsteil: In ihm können wieder beliebige (und beliebig viele) Deklarationen folgen wie Konstanten- oder Variablen-Deklarationen (auch wieder Prozedur-Deklarationen und sogar lokale Module, vgl. 1.7.2!).

3. `BEGIN`

4. einer Anweisungsfolge

5. `END <ProzedurName>;`
   wobei *<ProzedurName>* derselbe Name wie im Kopf sein muß.

Das entspricht also dem Schema

```
PROCEDURE <ProzedurName>;
<Deklarationen>
BEGIN
  <Anweisungen>
END <ProzedurName>;
```

Prozeduren werden im Deklarationsteil des Moduls (also vor dem eigentlichen Hauptprogramm) deklariert. Sie werden durch Nennung ihres Prozedurnamens in einer Anweisungsfolge aufgerufen. Dadurch wird die Abarbeitung des aufrufenden Programmstücks unterbrochen und mit der Prozedur fortgefahren. Ist diese beendet, geht es mit der Anweisung weiter, die dem Prozeduraufruf folgt. Das Schema lautet also:

```
MODULE <Modul-Name>;
CONST ...
TYPE ...
VAR ...

PROCEDURE p1;
  CONST ...
  TYPE ...
  VAR ...
BEGIN
    <Anweisungsfolge>
END p1;

PROCEDURE p2;
  CONST ...
  TYPE ...
  VAR ...
BEGIN
    <Anweisungsfolge>
END p2;
```

```
BEGIN   (* Hauptprogramm *)
  <Anweisungen>
  p1;
  <Anweisungen>
  p2;
  <Anweisungen>
  p1;
  <Anweisungen>
END <Modul-Name>.
```

Prozeduren haben also bis auf das Semikolon am Ende die gleiche Form wie der Hauptmodul (am Ende des Hauptmoduls steht ein Punkt).

Alle Bezeichner, die im Deklarationsteil einer Prozedur deklariert werden, sind nur innerhalb dieser Prozedur bekannt, also nur zwischen Prozedur-Kopf und dem dazugehörigen `END`. Ihr »Sichtbarkeitsbereich« (engl. *scope*) ist also auf die Prozedur beschränkt, in der sie definiert sind. Man nennt solche Bezeichner (Konstanten, Variablen, Typen, Prozeduren....) »lokal«. Die Bezeichner des Hauptprogramms sind dagegen »global«. Da letztere zwischen dem Modul-Kopf und dem Modul-Ende (also im gesamten Modul) sichtbar sind, sind diese auch in allen dort definierten Prozeduren gültig. Nur wenn ein Bezeichner an einer Stelle »sichtbar« (gleichbedeutend mit »definiert« oder »bekannt«) ist, kann er dort benutzt werden.

Folgendes muß man wissen:

- Prozeduren können mehrmals aufgerufen werden.
- Prozeduren können auch von anderen Prozeduren aufgerufen werden.
- Bei Single-pass-Compilern (das sind solche, die den Programmtext in nur einem einzigen Durchgang lesen, dazu gehören Megamax und SPC) müssen Prozeduren vor ihrem ersten Aufruf deklariert sein. Das bedeutet, eine Prozedur kann keine Prozedur aufrufen, die im Programmtext erst weiter unten definiert wird. Eine Ausnahme ist mit der `FORWARD`-Deklaration möglich (siehe unten).
- Eine Prozedur `p2` kann auch innerhalb einer anderen Prozedur `p1` deklariert sein. `p2` ist dann lokal zu `p1`. Die in `p1` definierten Bezeichner (Variablen etc.) sind dann »global« zu `p2` (aber »lokal« zu `p1`); sind also in `p2` sichtbar und können dort normal benutzt werden. Die Bezeicher des Hauptprogramms bleiben innerhalb `p2` natürlich sichtbar. Die innerhalb `p2` definierten Bezeichner (lokal zu `p2`) sind innerhalb `p1` nicht sichtbar und natürlich erst recht nicht im Hauptprogamm. `p2` ist vom Hauptprogramm aus ebenfalls nicht sichtbar und kann von dort aus nicht aufgerufen werden.
- Konstanten, Variablen oder Typen, die innerhalb einer Prozedur definiert werden (also »lokal« zu der Prozedur sind) dürfen den gleichen Namen tragen wie ein globaler Bezeich-

ner. Der globale Bezeichner ist dann innerhalb dieser Prozedur nicht mehr sichtbar, da der lokale Bezeichner Vorrang hat (sonst wäre die lokale Definition mit gleichem Namen wirkungslos). Der globale Bezeichner wird also durch einen lokalen Bezeichner mit demselben Namen »verdeckt«.

- Benutzen Sie möglichst viele lokale Variablen. Sie entlasten das Hauptprogramm, bringen Übersicht und machen die Prozedur unabhängiger vom übrigem Programm. Merken sie sich den Spruch: »Soviel lokal wie möglich, so wenig global wie nötig« (manche Zeitgenossen wandeln diese goldene Regel um in »Soviel ins Lokal wie möglich, so wenig nach draußen wie nötig...«).

Zur FORWARD-Deklaration

Falls eine Prozedur `p1` eine Prozedur `p2` aufruft, die ihrerseits `p1` aufruft, muß `p1` die Prozedur `p2` kennen und umgekehrt. Bei Single-pass-Compilern programmiert man dann:

```
FORWARD PROCEDURE p1;  (* Bei Megamax *)
(* PROCEDURE p1; FORWARD;  sonst *)

PROCEDURE p2;
BEGIN
  <...>
  p1;             (* p1 ist nach obiger FORWARD-Deklaration bekannt *)
  <...>
END p2;

PROCEDURE p1;
BEGIN
  <...>
  p2;
  <...>
END p1;
```

Eine andere Möglichkeit wäre es, `p2` lokal zu `p1` zu deklarieren:

```
PROCEDURE p1;

  PROCEDURE p2;
  BEGIN
    <...>
    p1;          (* p1 ist bekannt *)
    <...>
  END p2;
```

```
BEGIN
  <...>
  p2;               (* p2 ist bekannt *)

  <...>
END p1;
```

1.5.2 Prozeduren mit Parametern

Prozeduren sind besonders dann praktisch, wenn man dieselben Anweisungsfolgen immer wieder an verschiedenen Stellen benötigt. Statt jedesmal die Anweisungen auszuschreiben, packt man sie einmal in eine Prozedur und ruft an den entsprechenden Stellen nur die Prozedur auf.

Prozeduren können also Schreibarbeit ersparen und Codeverdopplung verhindern; gleichzeitig helfen sie, das Gesamtproblem in logisch zusammenhängende Einheiten zu gliedern.

Wir haben bereits davon in den Beispielen »`BitSetTest`« und »`BitMuster`« in Abschnitt 1.3.7 sowie »`milderLehrer`« im vorigen Abschnitt Gebrauch gemacht.

Während es im letzten Beispiel nur um Schreibersparnis ging, zeigt das erste Beispiel, daß Prozeduren auch Daten vom Hauptprogramm übermittelt bekommen können. Das ist nötig, wenn man zwar immer wieder dieselben Anweisungen benötigt, aber mit anderen Werten (»Parametern«) arbeitet. Dafür kann eine Prozedur eine »Parameterliste« erhalten; diese befindet sich im Prozedurkopf hinter dem Prozedurnamen. Sie enthält die Variablennamen (und deren Typen), in denen der Prozedur die Parameterwerte mitgegeben werden.

Einige fertige Prozeduren sind bereits bekannt, wie `WriteString` aus dem Modul `InOut`, die eine Zeichenkette als Parameter besitzt. Nun lernen Sie selbst Prozeduren zu schreiben. Damit können sie quasi den Sprachumfang selbst erweitern!

Als Beispiel dient die folgende Prozedur, die das Minimum zweier `CARDINAL`-Zahlen ausgibt:

```
MODULE ProzedurDemo1;

FROM InOut IMPORT  WriteCard, Read;

PROCEDURE Minimum(n, m : CARDINAL);
BEGIN
  IF n < m THEN WriteCard(n,1) ELSE WriteCard(m,1) END
END Minimum;
```

```
VAR i, j  : CARDINAL;
    taste : CHAR;

BEGIN
  i := 30; j := 50;
  Minimum(i,j);
  Read(taste)
END ProzedurDemo1.
```

Der Prozedur `Minimum` werden also die Werte `30` und `50` »übergeben«. Man spricht von »Werteparametern«.

Intern geschieht das so:

Beim Aufruf von `Minimum(i,j)` werden die Zahlen 30 und 50, also die Werte der Variablen `i` und `j`, in einen gesonderten Speicherbereich kopiert. Normalerweise ist dies der sogenannte Stack (»Stapel«). Dann springt die Programm-Ausführung zur Prozedur `Minimum`. Anhand der Parameterliste weiß die Prozedur, daß zwei Parameter und zwar `CARDINAL`-Zahlen vom Stack »abzuholen« sind.

Mit diesen Parametern arbeitet die Prozedur. Falls die Prozedur diese verändert, passiert den Variablen `i` und `j` aus dem Hauptprogramm nichts, da die Prozedur ja nur mit einer Kopie von deren Werten arbeitet.

Dazu ein weiteres Beispiel:

```
MODULE ProzedurDemo2;

FROM InOut IMPORT  WriteCard, WriteLn, Read;

VAR i : CARDINAL;

PROCEDURE verdoppeln(n : CARDINAL);
BEGIN
  n := 2*n;
  WriteLn; WriteCard(n,1)
END verdoppeln;

VAR taste : CHAR;

BEGIN
  i := 100;
  verdoppeln(i);
```

```
  WriteLn; WriteCard(i,1);
  Read(Taste)
  END ProzedurDemo2.
```

Die Prozedur `WriteCard` in `verdoppeln` gibt 200 aus, im Hauptprogramm 100. Die Variable `i` wurde also im Hauptprogramm nicht verändert.

Oft will man aber gerade eine solche Veränderung erreichen. Wendet man die Standardprozedur `INC` beispielsweise auf eine `CARDINAL`-Variable an, so wird ihr Wert erhöht, also verändert. Nach der Sequenz

```
n := 5;
INC(n);
```

ist also `n` zu `6` verändert worden. Dieses Verhalten erreicht man durch Verwendung eines »VAR-Parameters«. Dazu steht das Schlüsselwort `VAR` in der Parameterliste:

```
PROCEDURE verdoppeln(VAR n : CARDINAL);
```

Wenn wir das auf unser Beispiel anwenden:

```
MODULE ProzedurDemo3;

FROM InOut IMPORT  WriteCard, WriteLn, Read;

PROCEDURE verdoppeln(VAR n : CARDINAL);
BEGIN
  n := 2*n;
  WriteLn; WriteCard(n,1)
END verdoppeln;

VAR i     : CARDINAL;
    taste : CHAR;

BEGIN
  i := 100;
  WriteLn; WriteCard(i,1);
  verdoppeln(i);
  WriteLn; WriteCard(i,1);
  Read(taste)
END ProzedurDemo3.
```

Nun erzeugt es die Ausgabe

```
100
200
200
```

ist also nach Rückkehr von `verdoppeln` im Hauptprogramm verändert. Dies macht der Rechner so: Nicht der Wert, sondern die Adresse (das ist die Nummer der Speicherstelle, an der sich die betreffende Variable befindet) wird der Prozedur übergeben. Diese Adresse holt sich die Prozedur bei ihrem Aufruf. Sie arbeitet dann mit der betreffenden Speicheradresse und kann damit den Wert der Variablen des Hauptprogramms manipulieren.

Die Skizze verdeutlicht die verschiedene Arbeitsweisen:

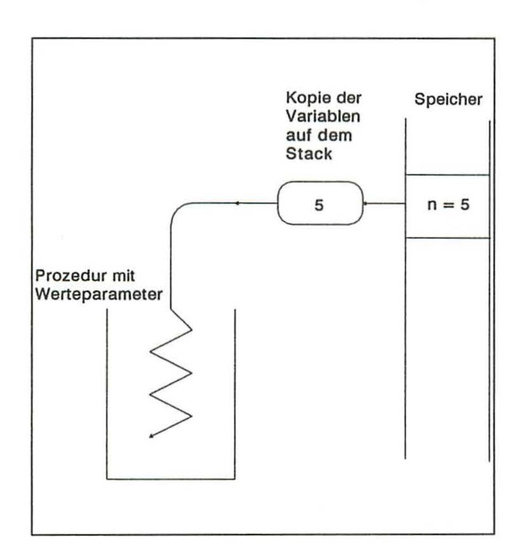


Bild 1.13a: Prozedur mit Werteparameter

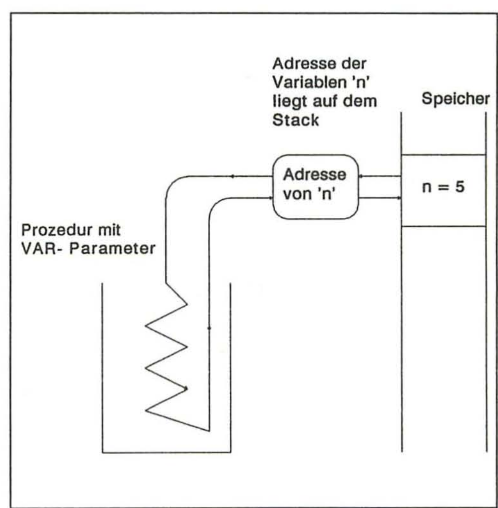


Bild 1.13b: Prozedur mit VAR-Parameter

Eine Prozedur mit VAR-Parametern kann aus diesem Grund nicht mit Konstanten aufgerufen werden. Beispielsweise würde `verdoppeln(5)` beim Kompilieren einen Fehler erzeugen.

Eine Prozedur kann mehrere Parameter haben, es können dabei Werte und VAR-Parameter gemischt auftreten. Ein Beispiel dafür liefert die Standardprozedur `INC` in der Version mit zwei Parametern. Deren Definition kann man sich – hier für `CARDINAL`-Variablen – so vorstellen:

```
PROCEDURE INC(VAR x: CARDINAL; n: CARDINAL);
```

Durch den Aufruf von `INC(i,5)` wird `i` um den Wert 5 erhöht.

In der Prozedur

```
PROCEDURE pl (x,y:CARDINAL; VAR u,v:INTEGER; ok:BOOLEAN);
```

sind `x`, `u`, `ok`, Werteparameter und `u`, `v` VAR-Parameter. Die folgende Prozedur berechnet den ggT und kgV zweier `CARDINAL`-Zahlen und gibt sie als VAR-Parameter zurück:

```
PROCEDURE kgVggT (a,b: CARDINAL; VAR kgV, ggT: CARDINAL);
VAR
    rest, ab: CARDINAL;     (* lokale Variablen *)
BEGIN
  ab := a * b;
  REPEAT
    rest := a MOD b;
    a := b;
    b := rest
  UNTIL rest = 0;
  ggT := a;
  kgV := ab DIV ggT
END kgVggT;
```

1.5.3 Funktionsprozeduren

Die Standardprozedur `CAP` liefert zu einem Kleinbuchstaben den entsprechenden Großbuchstaben. Mit `CAP(ch)` wird die Variable `ch` aber nicht verändert; den Großbuchstaben erhält man quasi aus dem Funktionsnamen `CAP`:

```
Gross:=CAP("a");
```

`CAP` nennt man daher eine »Funktions-Prozedur« oder kurz »Funktion«.

Der Computer macht das so: Das Ergebnis der Funktion wird nach ihrer Abarbeitung an einer besonderen Stelle abgelegt, zum Beispiel in einem Register des 68000er Prozessors oder auf einem Stack (bei Megamax, SPC). Bei der Wertzuweisung `Gross:=CAP("a")` erhält `Gross` diesen Wert.

Eine Funktion kann immer nur **ein** Ergebnis haben. Sie können also im Beispiel des letzten Abschnittes nicht kgV und ggT mit einer Funktion zurückgeben. Der Kopf einer Funktion sieht so aus:

```
PROCEDURE <FunktionsName> : <Typ-Bezeichner>;
```

oder (mit Parametern)

`PROCEDURE <FunktionsName> <ParameterListe> : <Typ-Bezeichner>;`

Als Beispiel eine Funktion, die das Minimum zweier Zahlen zurückgibt:

```
PROCEDURE Minimum(a,b: CARDINAL): CARDINAL;
BEGIN
  IF a < b THEN RETURN a
          ELSE RETURN b
  END
END Minimum.
```

Die Anweisung `i:=Minimum(3,5)` ergibt `i=3`.

Das Beispiel `BitSetTest` in Abschnitt 1.3.7 enthält vier Funktionen vom Typ Word.

Wichtig:

- Funktionen liefern ein Ergebnis. Dieses muß auch »gelesen« werden, zum Beispiel

`i:=Minimum(3,5);`	durch Zuweisen an eine Variable
`i:=3*Minimum(3,5);`	durch Weiterverarbeiten
`WriteCard( Minimum(3,5),2);`	durch Übergabe an eine Prozedur.

  Nicht erlaubt ist einfach das Aufrufen wie eine Prozedur:

`Minimum(3,5);`	Der Compiler weiß hier nicht, wohin mit dem Ergebnis.

- Der Wert, den eine Funktion als Ergebnis zurückgeben soll, wird hinter das Schlüsselwort `RETURN` geschrieben. Durch `RETURN 5;` wird zum Beispiel `5` zurückgeliefert (die Funktion »erhält den Wert« 5). Gleichzeitig wird die Funktion bei der RETURN-Anweisung beendet. Eine Funktion muß mindestens eine RETURN-Anweisung enthalten und durch eine RETURN-Anweisung beendet werden (also nicht mit `IF` eventuell daran vorbeilaufen), sonst bleibt der Funktionswert undefiniert.

- Beim Aufruf von Funktionen, die keine Parameterliste haben, muß hinter dem Namen dennoch eine »leere« Parameter-Liste »`()`« angegeben werden:

  ```
  x := ergebnis();
  ```
  Falsch ist hier
  ```
  x := ergebnis;
  ```

  da diese Anweisung eine andere Bedeutung hat, auf die wir in 1.6.7 noch eingehen.

Der korrekte Umgang mit Funktionen und Prozeduren ist sicherlich für den Anfänger ein harter Brocken. Sie können sich an den folgenden – nicht ganz fehlerfreien! – (Gegen-)Beispielen selbst prüfen. Angenommen, Sie wollen eine Routine `Low` schreiben, die ähnlich wie

CAP arbeitet und zu einem Großbuchstaben den entsprechenden Kleinbuchstaben liefert, andere Zeichen aber nicht verändert (also das Gegenteil von CAP).

Analysieren Sie die folgenden Versionen, die alle ordnungsgemäß kompiliert werden! Noch ein Hinweis. Kleinbuchstaben haben einen um 32 erhöhten ASCII-Wert wie der entsprechende Großbuchstabe:
CHR(ORD("A") + 32) liefert "a".

```
PROCEDURE Low1(ch: CHAR);
BEGIN
  IF ("A" <= ch) AND (ch <= "Z") THEN
     ch := CHR(ORD(ch) + 32))
  END
END Low1;

PROCEDURE Low2(VAR ch: CHAR);
BEGIN
  IF ("A" <= ch) AND (ch <= "Z") THEN
     ch := CHR(ORD(ch) + 32))
  END
END Low2;

PROCEDURE Low3(ch: CHAR): CHAR;
BEGIN
  IF ("A" <= ch) AND (ch <= "Z") THEN
     RETURN CHR(ORD(ch) + 32))
  END
END Low3;

PROCEDURE Low4(ch: CHAR): CHAR;
BEGIN
  IF ("A" <= ch) AND (ch <= "Z") THEN
     RETURN CHR(ORD(ch) + 32))
  ELSE
     RETURN ch
  END
END Low4;

PROCEDURE Low5(VAR ch: CHAR): CHAR;
BEGIN
  IF ("A" <= ch) AND (ch <= "Z") THEN
     RETURN CHR(ORD(ch) + 32))
  ELSE
```

```
      RETURN ch
    END
  END Low5;

  PROCEDURE Low6(VAR ch: CHAR): CHAR;
  BEGIN
    IF ("A" <= ch) AND (ch <= "Z") THEN
      ch := CHR(ORD(ch) + 32))
    END;
    RETURN ch
  END Low6;
```

Wie sind diese Versionen zu bewerten? `Low1` ist völlig unbrauchbar, da das Zeichen nicht als Funktionswert zurückgegeben wird. Da hier `ch` kein VAR-Parameter ist, bewirkt `Low1` überhaupt nichts.

`Low2` macht zumindest etwas Sinnvolles: es wandelt das eingegebene Zeichen in einen Kleinbuchstaben; das geht, weil hier ein VAR-Parameter benutzt wurde. Aber `Low2` soll ja wie `CAP` arbeiten, also den Eingabewert unverändert lassen und den Kleinbuchstaben nur als Wert zurückliefern, sie muß daher als Funktion deklariert werden. Ein Aufruf wie

```
Write(Low2(ch));
```

ist also unmöglich, man könnte sich bestenfalls mit

```
Low2(ch);
Write(ch);
```

behelfen.

`Low3` ist fehlerhaft. Falls das übergebene Zeichen kein Großbuchstabe ist, erfolgt keine Ergebniszuweisung (da dann kein `RETURN` aufgerufen wird). Der Compiler kann solche Fehler leider nicht melden, da er nicht so leicht feststellen kann, welche Fälle beim Ablauf auftreten können. Manche Systeme bemerken den Fehler aber während der Laufzeit!

Die Funktion `Low4` erfüllt die gestellten Anforderungen. Sie enthält zwei RETURN-Anweisungen; eine im THEN-Fall und eine im ELSE-Fall. Damit ist sichergestellt, daß immer (in jedem »Fall«) ein definierter Wert zurückgeliefert wird. Wenn das eingegebene Zeichen kein Großbuchstabe war, soll schließlich der urspüngliche Wert zurückgegeben werden.

`Low5` arbeitet ebenfalls korrekt, hat jedoch einen Haken. Die Deklaration des Parameters als VAR-Parameter ist überfüssig, da `ch` in der Funktion überhaupt nicht verändert wird. Außerdem verhindert das, daß Konstanten übergeben werden können:

```
klein := Low5("A");
```

wäre also nicht möglich.

Die Funktion `Low6` scheint richtig zu arbeiten, hat aber einen kleinen »Nebeneffekt«: Es verändert die übergebene Variable gleich mit, da `ch` hier als VAR-Parameter deklariert ist. Wie bei `Low5` sind hier keine Konstanten als Argumente möglich. Läßt man das `VAR` weg, läuft die Funktion fehlerfrei.

Zum Schluß noch eine endgültige Version, die auch die Umlaute (Ä, Ö, Ü) richtig umwandelt:

```
PROCEDURE Low(ch: CHAR): CHAR;

BEGIN
  CASE ch OF
    "A".."Z" : RETURN CHR(ORD(ch) + 32)) |
    "Ä"      : RETURN "ä" |
    "Ö"      : RETURN "ö" |
    "Ü"      : RETURN "ü"
    ELSE RETURN ch
  END
END Low;
```

Die verschiedenen Versionen sind auf der Diskette im Modul »`PROZDEM4.M`« enthalten.

`RETURN` darf auch in Prozeduren, die keine Funktionen sind, zum vorzeitigem Abbruch benutzt werden. Dahinter darf dann jedoch kein Ausdruck stehen, da reine Prozeduren keinen Rückgabewert besitzen. Das folgende Beispiel finden sie auch auf der Diskette (Filename: »`RETURN.M`«:)

```
PROCEDURE VorzeitigerAbbruch;

VAR i: CARDINAL;

BEGIN
  FOR i := 0 TO 1000 DO
    IF i = 30 THEN RETURN END;
    WriteLn; WriteCard(i,1);
  END
END VorzeitigerAbbruch;
```

Die Prozedur schreibt die Zahlen von 0 bis 29. Zugegeben, in diesem Beispiel ist der »brutale Ausstieg« aus Schleife und Prozedur nicht gerade die feine englische Art...

1.5.4 Rekursion

Vielleicht kennen Sie die Werbung, in der ein Fernsehgerät gezeigt wird, in dem ein Fernsehgerät gezeigt wird, auf dem wiederum ein Fernsehgerät gezeigt wird ...

Dieser Effekt heißt »Bild im Bild« und läßt sich einfach dadurch realisieren, indem man eine Videokamera an den Fernseher anschließt (eventuell über einen Videorecorder als Adapter) und die Kamera dann auf den Fernseher (der das Kamera-Bild zeigen sollte) richtet.

In Modula gehört so ein »Bild im Bild«-Effekt zu den sehr effizienten Programmierwerkzeugen. Zunächst betrachten wir einen normalen Aufruf zweier Prozeduren:

Gegeben sei eine Prozedur `A`, die von einer Prozedur `B` aufgerufen wird, `B` wird vom Hauptprogramm aus aufgerufen. Das Hauptprogramm wird dann beim Prozeduraufruf von `B` unterbrochen; der Ablauf bei `B` weitergeführt.

```
MODULE M;

PROCEDURE A;
BEGIN
  ...
END A;

PROCEDURE B;
BEGIN
  ...
  A;
  ...
END B;

BEGIN    (* M *)
  ...
  B;
  ...
END M.
```

Beim Aufruf von `A` wird `B` unterbrochen, und es folgt die Abarbeitung von `A`. Nach getaner Tat wird der Rest von `B` erledigt. Genauer werden diejenigen Anweisungen, die dem Prozeduraufruf von `A` folgen, ausgeführt. Ist auch das geschehen, wird der Rest des Hauptprogramms erledigt.

Was passiert aber, wenn `A` und `B` identisch sind, das heißt, wenn `B` sich selbst aufruft? Analysieren wir das folgende Beispiel:

```
MODULE RekursionsTest1;

FROM InOut IMPORT Read, Write, WriteString, WriteLn;

PROCEDURE Zeichen;
VAR ch : CHAR;
BEGIN
  Read(ch);
  IF ch # "." THEN Zeichen END;
  Write(ch)
END Zeichen;

VAR taste : CHAR;

BEGIN
  WriteString("Geben Sie einige Zeichen ein, Punkt zum Abschluß!");
  WriteLn;
  Zeichen;
  Read(taste)
END RekursionsTest1.
```

Gibt man nacheinander »123.« ein, so erhält man auf dem Bildschirm

```
123..321
```

Wieso?

Das Hauptprogramm ruft die Prozedur `Zeichen` auf, die erste Anweisung lautet `Read(ch)`. Die eingegebene `1` wird auf dem Bildschirm »geechot«. Da `1` ungleich ».« ist, wird wiederum `Zeichen` aufgerufen. Der Rechner »merkt« sich aber, daß von der Prozedur `Zeichen` vom ersten Aufruf noch ein Rest abzuarbeiten ist. Nun geben wir eine `2` ein. Wiederum wird `Zeichen` aufgerufen, da `2` ungleich ».« ist. Wir geben noch »3« und ».« in gleicher Weise ein. Auf dem Bildschirm steht bis jetzt:

```
123.
```

Nun erfolgt die Abarbeitung des Prozedurrestes. `Write(ch)` schreibt also das letzte eingegebene Zeichen, also ».« noch einmal auf den Bildschirm. Dieser letzte Prozedur-Aufruf ist jetzt beendet. Nun ist aber noch der Rest vom dritten Aufruf abzuarbeiten. Hier war `ch`= 3, also wird als nächstes eine `3` aufgeschrieben. Der Rest des zweiten Aufrufes erzeugt die `2`, der Rest des ersten Aufrufes die `1`. Somit sind alle Prozeduraufrufe abgearbeitet, und der Rest des Hauptprogramms folgt.

Zum besseren Verständnis beachte man, daß für eine lokale Variable einer Prozedur (hier `ch`) bei jedem Aufruf der Prozedur Speicherplatz bereitgestellt wird. Der Wert einer lokalen Variablen wird erst dann »vergessen«, wenn die Prozedur verlassen wird. Daher kennt `Zeichen` zu guter Letzt auch noch die abschließende `1`. Ändern Sie das Programmm einmal ab, indem Sie die Zeile `VAR ch: CHAR;` vor die Prozedur `Zeichen` setzen, also als globale Variable deklarieren. Finden Sie heraus, was nun bei der obigen Eingabe passiert!

Diesen »Selbstaufruf« einer Prozedur nennt man ***Rekursion***. Die Skizze macht das Verfahren deutlich:

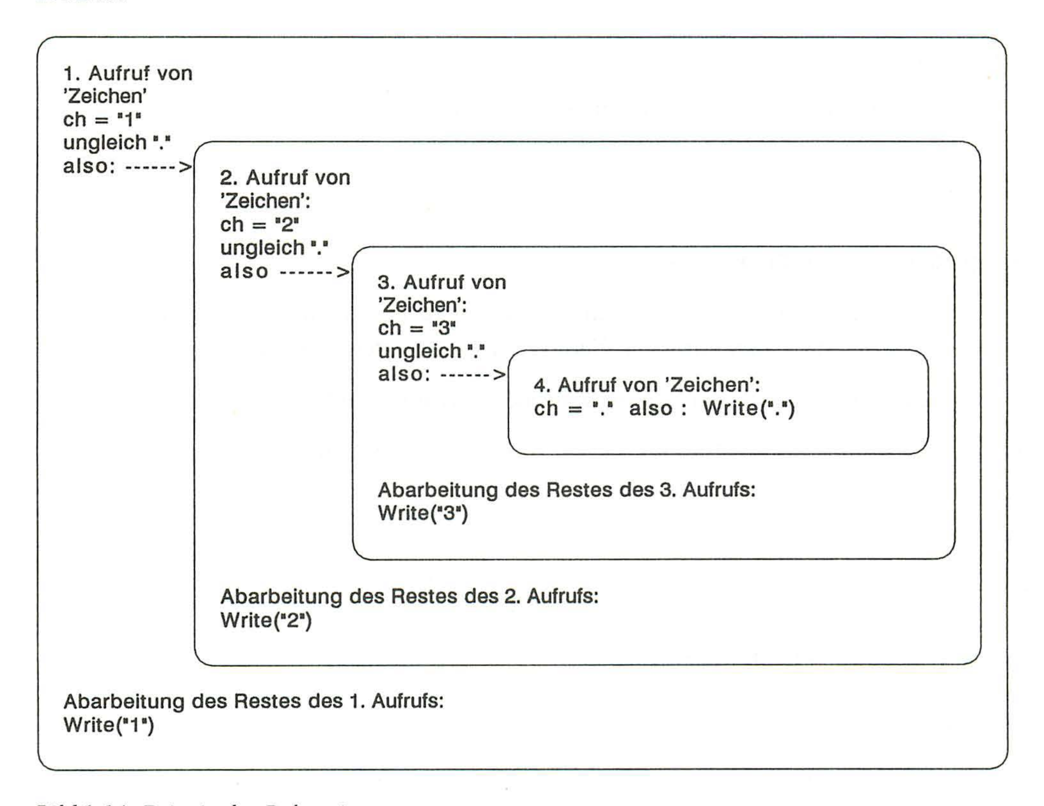


Bild 1.14: Prinzip der Rekursion

Eine Kette von Selbstaufrufen muß natürlich einmal abbrechen, hierzu dient die IF-Anweisung! Diese Abbruchsbedingung ist also ein wesentlicher Bestandteil von rekursiven Prozeduren! Sie kann in der Mitte der Prozedur liegen:

```
PROCEDURE rek;
BEGIN
  <Anweisungsfolge vor Rekursion>
  IF <Bedingung> THEN rek END;
  <Anweisungsfolge nach Rekursion>
END rek;
```

Damit erhält man die Abarbeitungsreihenfolge:

Av_1, Av_2, Av_3, ..., Av_n (Bedingung falsch) An_n, ..., An_3, An_2, An_1

dabei bedeutet

Av_i = Anweisungsfolge vor Rekursion im i-ten Aufruf und
An_i = Anweisungsfolge nach Rekursion im i-ten Aufruf

`Av` oder `An` können auch leer sein

Beispiele rekursiver Funktionen

Rekursion ist ein machtvolles Programmierwerkzeug, da mit ihr Sachverhalte, die »rekursiven Charakter« haben, in natürlicher Weise formuliert werden können.

Wir greifen einige Beispiele der vorangegangenen Abschnitte auf:

Zur `ggT`-Berechnung (vgl. Modul `DrittesBeispiel` aus Abschnitt 1.1):

Es gilt:

```
(1) ggT(i,j) = i,                falls i = j
(2) ggT(i,j) = ggT(i - j,j),     falls i > j
(3) ggT(i,j) = ggT(i,j - i),     falls i < j
```

Das setzt man wie folgt in Modula um:

```
MODULE RekursionsTest2;

FROM InOut IMPORT ReadCard, WriteCard, WriteString, WriteLn;

PROCEDURE ggT(a,b : CARDINAL) : CARDINAL;
BEGIN
  IF a = b THEN RETURN a
    ELSIF a > b THEN RETURN ggT(a-b,b)
    ELSE RETURN ggT(a,b-a)
  END
END ggT;
```

```
VAR i,j   : CARDINAL;

BEGIN
  REPEAT
    WriteString("Geben Sie eine natürliche Zahl ein : "); ReadCard(i);
    WriteString("Noch eine, bitte ( 1 = Ende )       : "); ReadCard(j);
    WriteLn;
    WriteString("Der ggT dieser beiden Zahlen lautet: ");
    WriteCard(ggT(i,j),1);
    WriteLn
  UNTIL j = 1
END RekursionsTest2.
```

Bei diesem Beispiel gibt es keine eigentlichen Anweisungsfolgen vor oder nach der Rekursion (bzw. sie sind leer). Der Algorithmus stimmt trotzdem, da die Zahlen `a` und `b` dauernd durch die Aufrufe in der Rekursion erniedrigt werden.

Vor einem Prozeduraufruf muß im Computer für die gerufene Prozedur Speicherplatz für lokale Variablen und Werteparameter der Prozedur angelegt werden. Dann erst wird sie betreten. Dies nennt man dann eine *Inkarnation* (dt.: etwa »Fleischwerdung«). Ruft sich eine Prozedur selbst auf, so hat man eine neue Inkarnation dieser Prozedur. Jede Inkarnation hat ihre eigenen lokalen Variablen und Parameter. Insofern sind auch die Werte `a` und `b` in jeder Inkarnation verschieden! Globale Variablen und VAR-Parameter werden natürlich nicht neu angelegt, ihre Werte werden gegebenenfalls überschrieben.

Ein weiteres klassisches Beispiel ist die Fakultätsberechnung (vgl. auch Abschnitt 1.3.2). Es gilt die rekursive mathematische Definition:

1. `0! = 1` sowie
2. `n! = n * (n-1)!` für n > 0

Dies spiegelt sich in der rekursiven Fassung der Prozedur `Fakultaet` wieder.

```
MODULE RekursionsTest3;

FROM InOut IMPORT ReadCard, WriteCard, WriteString, WriteLn;

PROCEDURE Fakultaet(n : CARDINAL) : CARDINAL;
BEGIN
  IF n <= 1 THEN RETURN 1 ELSE RETURN n*Fakultaet(n-1) END
END Fakultaet;
```

```
VAR i : CARDINAL;

BEGIN
  REPEAT
    WriteString("Geben Sie eine Zahl (0 <= n <= 8 ) ein ( 0 = Ende): ");
    ReadCard(i); WriteLn;
    WriteString("Es gilt: "); WriteCard(i,1); WriteString("! = ");
    WriteCard(Fakultaet(i),1); WriteLn
  UNTIL i = 0
END RekursionsTest3.
```

Berechnen Sie `Fakultaet(5)`. Es sollte `120` herauskommen. Betrachtet man die Prozedur `Fakultaet`, so hat es den Anschein, als könne die Multiplikation nicht ausgeführt werden, weil der zweite Faktor `Fakultaet(n-1)` ja noch gar nicht berechnet worden ist. Der Rechner organisiert das folgendermaßen: Bei jedem Aufruf von `Fakultaet(n-1)` merkt der Rechner sich die Stelle, wo er bei `n*...` aufgehört hat. Ist durch die mehrfache Erniedrigung von n um eins schließlich 1 erreicht, so ist der Wert `Fakultaet(1)` bekannt, und die Multiplikation `2*1` kann sofort ausgeführt werden. Es folgt dann der Rücksprung zu der Multiplikation von `3*`, der Rechner rechnet also `3*2*1` usw. bis schließlich alle Aufrufe abgearbeitet sind und das Ergebnis `5*4*3*2*1=120` feststeht. Insgesamt hat sich die Prozedur hier fünfmal selbst aufgerufen. Man sagt, die *Rekursionstiefe* ist 5.

Im nächsten Beispiel wird die Nullstelle der Funktion

```
f(x) = x -cos(x), x im Bogenmaß (»Radiant«)
```

im Intervall `[a,b] = [0,1]` rekursiv ermittelt. Weil

```
f(0)<0 und f(1)>0
```

gilt, wird geprüft, ob der Funktionswert in der Intervallmitte positiv oder negativ ist. Im ersten Fall wird im Intervall `[a, (a+b)/2]` im zweiten Fall im Intervall `[(a+b)/2, b]` weitergesucht. Diese Berechnung wird solange fortgeführt, bis die Länge des Suchintervalls einen gewissen Wert unterschritten hat.

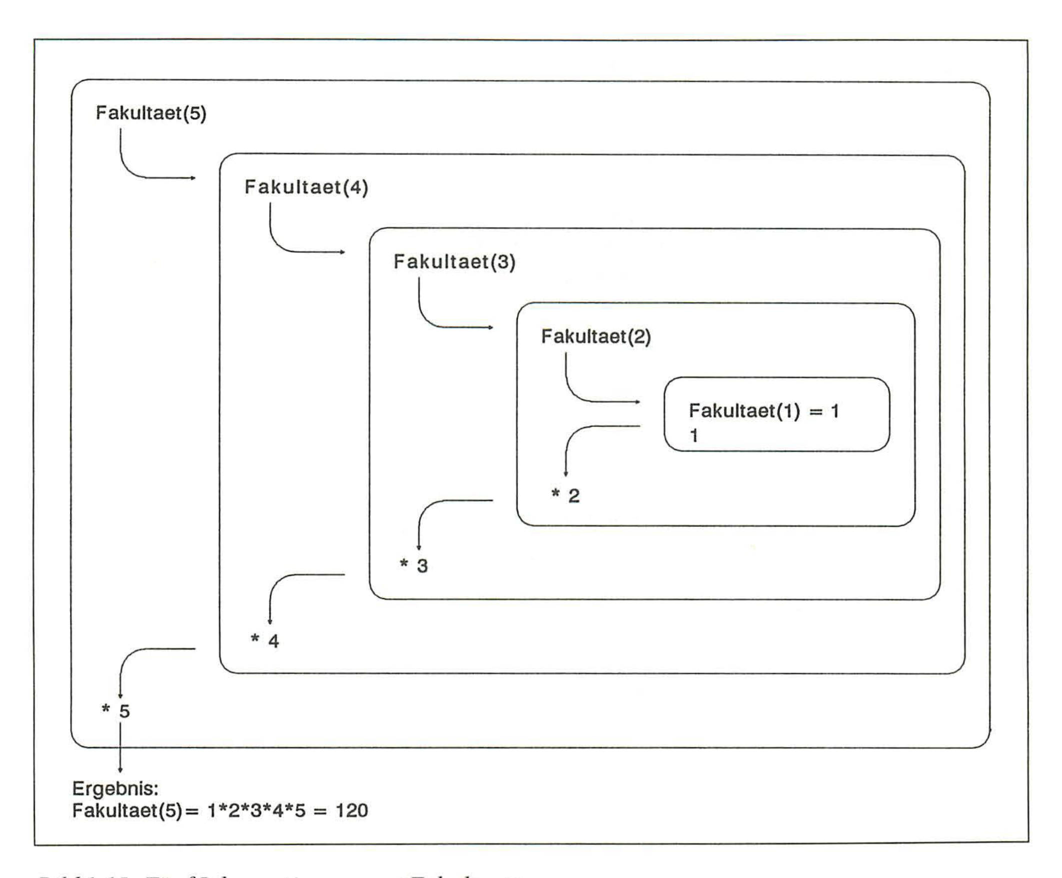


Bild 1.15: Fünf Inkarnationen von »Fakultaet«

```
MODULE RekursionsTest4;

FROM InOut    IMPORT WriteReal, WriteString, Read;
FROM MathLib0 IMPORT cos;

CONST epsilon = 1.0E-10;

PROCEDURE f(x : REAL) : REAL;
BEGIN
  RETURN x - cos(x)
END f;
```

```
PROCEDURE Nullstelle(VAR a,b : REAL) : REAL;

VAR mitte : REAL;

BEGIN
  mitte := (a + b) / 2.0;
  IF ABS(a-b) < epsilon THEN RETURN mitte
    ELSIF f(mitte) < 0.0 THEN RETURN Nullstelle(mitte,b)
    ELSE RETURN Nullstelle(a,mitte) END
END Nullstelle;

VAR x1,x2 : REAL;
    taste : CHAR;

BEGIN
  x1 := 0.0; x2 := 1.0;
  WriteString("Die Nullstelle von 'f(x) = x - cos(x)' lautet: ");
  WriteReal(Nullstelle(x1,x2),12,10);
  Read(taste)
END RekursionsTest4.
```

Man erhält das Ergebnis `0.7390851332`.

Die Prozedur `Nullstelle` läßt sich natürlich auch »iterativ« (also nicht rekursiv, sondern mit einer Schleife) fassen:

```
PROCEDURE Nullstelle(a,b: REAL): REAL;

VAR mitte: REAL;

BEGIN
  REPEAT
    mitte := (a + b) / 2.0;
    IF f(mitte) < 0.0
      THEN a := mitte
      ELSE b := mitte
    END;
  UNTIL ABS(a-b) < Toleranz:
  RETURN mitte
END Nullstelle;
```

Das komplette Programm ist auf der Diskette unter dem Namen »`NULLITER.M`« vorhanden.

Die Schwächen von Rekursionen

Wir bringen ein weiteres klassisches Beispiel:

Leonardo von Pisa, genannt *Fibonacci,* formulierte in seinem 1202 erschienenen Buch »liber abaci« die berühmte Kaninchenaufgabe:

»Zur Zeit 0 hat man ein Kaninchenpaar. Nach 2 Monaten erzeugt dieses Kaninchenpaar jeden Monat ein neues. Die Nachkommen befolgen auch dieses Gesetz. Wieviele Paare hat man nach n Monaten?«

Bezeichnet man die Anzahl der Kaninchen zur Zeit `n` mit `Fibonacci(n)`, so gilt offenbar:

```
(1)   Fibonacci(0) = 1
(2)   Fibonacci(1) = 1
```

und

```
(3)   Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)
```

↑ Paare, die schon da sind (gestorben wird nicht).

↑ Kaninchenpaare, die schon mindestens zwei Monate alt sind, haben ein neues Paar erzeugt.

Die jeweils nächste Zahl erhält man also einfach durch Addition der letzten beiden:

`1,1,2,3,5,8,13,21,34,55,89,...`

In der folgenden Realisation messen wir die Rechenzeiten mit einer »Stoppuhr«, die wir in Kapitel 3 erläutern.

Vor jeder Berechnung wird die Uhr durch `Stoppuhr.Start` auf Null gestellt, nach jeder Berechnung liest man die Zeit in Millisekunden mittels `Stoppuhr.Lesen()`. Unsere Stoppuhr hat eine Auflösung von 5 ms.

Wir benutzen hier zum erstenmal einen Modul in der Importliste, der nicht vom Megamax-System mitgeliefert wurde, sondern von uns stammt. Vor der Übersetzung des Programms muß dem System die Übersetzung des Moduls `Stoppuhr` bekannt sein! Diese Übersetzung finden Sie in dem Ordner »`OBJEKTE`«. Bevor Sie mit der Kompilation von »`REKUTES5.M`« beginnen, muß dem Compiler der Suchpfad für »`OBJEKTE`« bekannt gemacht werden. Hierzu ist die Datei »`SHELL.INF`« entsprechend zu edieren (vgl. Handbuch). Wenn sie nicht das Megamax-System benutzen, streichen Sie einfach die Stoppuhr-Aufrufe.

```
MODULE RekursionsTest5;

FROM InOut  IMPORT WriteCard, WriteString, WriteLn, Read, RedirectOutput;

IMPORT Stoppuhr;

PROCEDURE Fibonacci(n : CARDINAL) : CARDINAL;
BEGIN
  IF n <= 1 THEN RETURN 1
    ELSE RETURN Fibonacci(n-1) + Fibonacci(n-2) END
END Fibonacci;

VAR i, fib : CARDINAL;
    zeit   : LONGCARD;
    taste  : CHAR;

BEGIN
  RedirectOutput("PRN:",FALSE);  (*Streichen wenn kein Drucker vorhanden *)
  WriteString("  i | Fibonacci(i) | Zeit in Millisekunden"); WriteLn;
  WriteString("----------------------------------------"); WriteLn;
  FOR i := 0 TO 23 DO
    WriteCard(i,3);    WriteString(" |");
    Stoppuhr.Start; fib:=Fibonacci(i); zeit := Stoppuhr.Lesen();
    WriteCard(fib,10); WriteString("     |"); WriteCard(zeit,13); WriteLn
  END;
  Read(taste)
END RekursionsTest5.
```

Die Ausgabe wird mit `InOut.RedirectOutput` auf den Drucker (»`PRN:`«, wie engl. *printer*=»Drucker«) umgeleitet. Falls kein Drucker angeschlossen ist, muß dieser Prozeduraufruf unterbleiben!

i	Fibonacci(i)	Zeit in Millisekunden
0	1	0
1	1	0
2	2	0
3	3	0
4	5	0
5	8	0
6	13	0
7	21	5
8	34	5
9	55	5
10	89	10
11	144	10
12	233	15
13	377	30
14	610	45
15	987	80
16	1597	130
17	2584	205
18	4181	340
19	6765	545
20	10946	880
21	17711	1425
22	28657	2305
23	46368	3730

Die Stoppuhr hat eine Auflösung von 5 ms (Millisekunden). Man erkennt, daß auch die Rechenzeiten in etwa die Rekursions-Formel

```
Zeit(n) = Zeit(n-1) + Zeit(n-2)
```

befolgen. Woran liegt das? Der Aufruf von `Fibonacci(5)` erzeugt die folgenden Aufrufe, die wir in einem »Baum« darstellen:

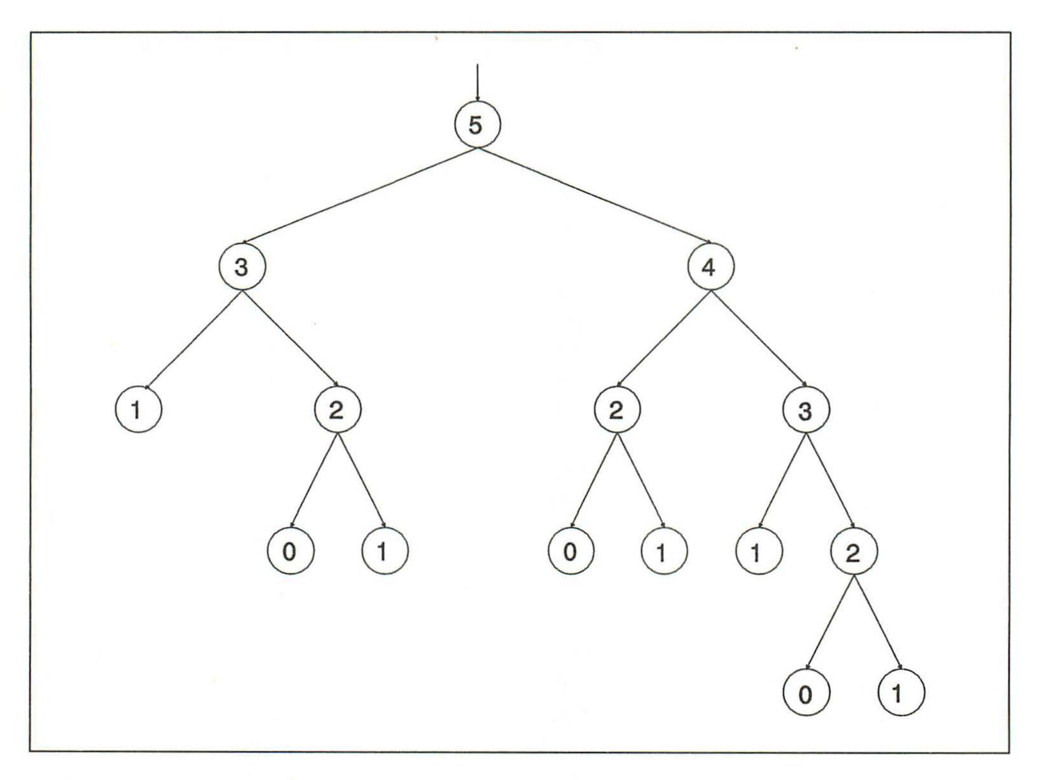


Bild 1.16: Fibonacci-Baum

Insgesamt wird `Fibonacci` hier schon 15 mal aufgerufen! Wir haben also bei einer Rekursionstiefe von nur 5 bereits 15 Inkarnationen von `Fibonacci`:

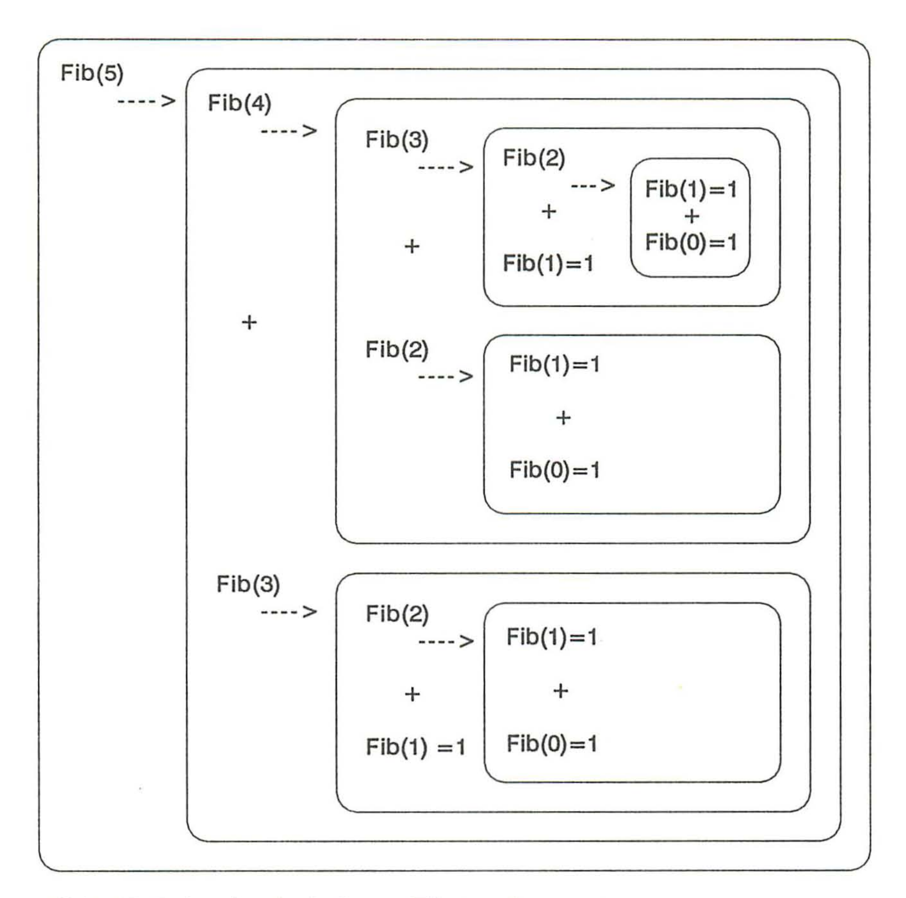


Bild 1.17: Rekursive Aufrufe von Fibonacci

Die iterative Version von Fibonacci ist etwas umständlicher. Sie erzeugt aber Rechenzeiten, die so kurz sind, daß sie mit unserer Stoppuhr nicht gemessen werden können, da sie unter 5 ms liegen:

```
MODULE FibonacciFolgeIterativ;

FROM InOut    IMPORT RedirectOutput, WriteCard, WriteString, WriteLn, Read;
FROM Stoppuhr IMPORT Start, Lesen;

PROCEDURE FibonacciIter(n : CARDINAL) : CARDINAL;
VAR i, neuer, letzter, vorletzter : CARDINAL;
BEGIN
```

```
  IF n <= 1 THEN RETURN 1 ELSE
    letzter := 1; vorletzter := 1;
    FOR i := 2 TO n DO
      neuer := letzter + vorletzter;
      vorletzter := letzter;
      letzter := neuer
    END;
    RETURN neuer
  END
END FibonacciIter;

VAR i, fib : CARDINAL;
    zeit   : LONGCARD;
    taste  : CHAR;

BEGIN
  RedirectOutput("PRN:",FALSE);
  WriteString("  i | Fibonacci(i) | Zeit in Millisekunden"); WriteLn;
  WriteString("-------------------------------------------"); WriteLn;
  FOR i := 0 TO 23 DO
    WriteCard(i,3);   WriteString(" |");
    Start; fib:=FibonacciIter(i); zeit := Lesen();
    WriteCard(fib,10); WriteString("     |"); WriteCard(zeit,13); WriteLn
  END;
  Read(taste)
END FibonacciFolgeIterativ.
```

Im Prinzip läßt sich zu jeder rekursiv formulierten Prozedur eine iterative Version angeben. Bei Prozeduren, die vor oder nach der Rekursion keine Anweisungsfolgen haben, ist das immer ohne Umstand möglich; andernfalls muß man ein wenig mit einer Stapelverwaltung tricksen. Wir gehen im Abschnitt 2.2.1 darauf ein. Im ersteren Fall sind die iterativen Fassungen immer deutlich schneller als die rekursive, da bei der Rekursion noch eine zeitaufwendige Speicherung von Rücksprungadressen, lokalen Variablen und Parametern (halt die Inkarnation) erfolgt.

Bei großer Rekursionstiefe können sehr viele Inkarnationen entstehen (siehe Fibonacci), die noch auf ihre Abarbeitung warten. Dabei kann irgendwann der Speicherplatz zu knapp werden. Das Programm bricht dann mit irgendeiner unschönen Meldung wie »Stacküberlauf« ab!

Warum überhaupt noch Rekursion?

Als Faustregel gilt: Immer dann, wenn das Problem in rekursiver Definition vorliegt, ist eine

rekursive Umsetzung leicht zu programmieren und vor allem leicht zu verstehen. Bei einer einfachen Rekursion, wo sich die rekursive Prozedur nur einmal selbst aufruft (bei `Fibonacci` war es zweimal!), wird die Abarbeitung nicht deutlich langsamer sein, und bei kleinen Rekursionstiefen kommt es wohl kaum zu einem Stack-Überlauf. Man sollte sich also als Programmierer über die mögliche Rekursionstiefe Klarheit verschaffen.

Rekursion ist insbesondere dann angesagt, wenn sich die Datenstruktur bereits rekursiv definieren läßt. Solche Datenstrukturen werden wir in Kapitel 2 betrachten. Dort folgen also noch ernsthafte Beispiele, die die vorteilhafte Anwendung von Rekursionen zeigen.

Beispiele mit großer Rekursionstiefe

Wir schließen das Kapitel mit zwei extremen Beispielen: Das erste Programm zeigt eine rekursive Definition der Potenzfunktion von ganzen Zahlen, die auf eine rekursiver Multiplikation und diese wieder auf eine rekursive Definition der Addition zurückgreift:

```
MODULE RekursionsTest6;

FROM InOut IMPORT ReadCard, WriteCard, WriteString, WriteLn;

PROCEDURE add(a,b : CARDINAL) : CARDINAL;
BEGIN
  IF b = 0 THEN RETURN a ELSE RETURN add(a,b-1) + 1 END
END add;

PROCEDURE mult(c,d : CARDINAL) : CARDINAL;
BEGIN
  IF d = 1 THEN RETURN c ELSE RETURN add(c,mult(c,d-1)) END
END mult;

PROCEDURE potenz(e,f : CARDINAL) : CARDINAL;
BEGIN
  IF f = 0 THEN RETURN 1 ELSE RETURN mult(e,potenz(e,f-1)) END
END potenz;

VAR n,m   : CARDINAL;
    taste : CHAR;

BEGIN
  REPEAT
    WriteString("Geben Sie eine nat. Zahl ein (0 = ENDE) : "); ReadCard(n);
    WriteString("Noch eine, bitte                       : "); ReadCard(m);
    WriteLn;
    WriteCard(n,1); WriteString(" hoch "); WriteCard(m,1);
```

```
    WriteString(" ergibt: "); WriteCard(potenz(n,m),1);
    WriteLn;
  UNTIL n = 0
END RekursionsTest6.
```

Wir meinen natürlich nicht, daß man so programmieren soll. Dieses Programm hat lediglich »didaktischen Wert«, bei größeren Zahlen merkt man wiederum die langsame Abarbeitung.

Im 2. Beispiel programmieren wir die sogenannte Ackermann-Funktion. Diese ist teuflisch rekursiv definiert:

(1) `ack(0,n) = n+1`
(2) `ack(m,0) = ack(m-1, 1)`
(3) `ack(m,n) = ack(m-1, ack(m,n-1))`

Der Mathematiker *W. Ackermann* fand sie bei der Fortsetzung der Folge

`Nachfolger` → `Summe` → `Produkt` → `Potenz`

Es gilt (was nicht leicht ersichtlich ist):

(a) `ack(1,n) = n + 2`
(b) `ack(2,n) = 2n + 3`
(c) `ack(3,n) =` $2^{n+3} - 3$

Sich mit diesen Sätzen einige Werte zum Testen auszurechnen ist deutlich einfacher als mit der eigentlichen Definition. Setzen Sie bitte bei dem Programm keine zu großen Zahlen ein! Die Ackermann-Funktion ist berüchtigt für große Rekursionstiefen und enorme Rechenzeiten, die sehr schnell mit der Größe der Eingaben wachsen!

```
MODULE AckermannFunktionDemo;

FROM InOut  IMPORT WriteCard, ReadLCard, WriteString, WriteLn;
IMPORT Stoppuhr;

PROCEDURE Ackermann(m,n : LONGCARD) : LONGCARD;
BEGIN
  IF m = 0D THEN  RETURN n+1D
    ELSIF n=0D THEN RETURN Ackermann(m-1D,1D)
  ELSE RETURN Ackermann(m-1D,Ackermann(m,n-1D)) END
END Ackermann;

VAR i,j   : LONGCARD;
```

```
BEGIN
  LOOP
    WriteString("i ( > 4 = ENDE) : "); ReadLCard(i);
    IF i > 4D THEN EXIT END;
    WriteString("j              : "); ReadLCard(j);
    WriteString("Ackermann(i,j)  = ");
    Stoppuhr.Start; WriteCard(Ackermann(i,j),1);
    WriteString("  Rechenzeit "); WriteCard(Stoppuhr.Lesen(),1);
    WriteString(" Millisekunden");  WriteLn;
  END
END AckermannFunktionDemo.
```

Die Ackermannfunktion wird oft von Studenten benutzt, um die Absturzsicherheit von Großrechenanlagen und die Geduld des Operators zu testen (nach dem Motto: »Wie schnell werde ich meine Benutzernummer los?«).

1.6 Selbstdefinierte Datentypen

Typen gibt's...

Im Gegensatz zu einfacheren Sprachen bietet Modula höchst flexible Gestaltungsmöglichkeiten, die in Abschnitt 1.3 besprochenen Standard-Datentypen zu komplexen Strukturen zu kombinieren. In welchen Fällen man diese Strukturen einsetzt, wie man sie definiert und handhabt, davon soll in diesem Abschnitt die Rede sein.

Zunächst geht es aber noch ergänzend zu Abschnitt 1.3 um Aufzählungstypen und Unterbereichstypen, die man noch zu den einfachen Typen zählt. Aber auch sie sind schon vom Benutzer zu deklarieren. Eine solche Vereinbarung geschieht in einer »Typ-Deklaration«. Sie hat die Form

```
TYPE
    <Bezeichner1> = <Typ1>;
    <Bezeichner2> = <Typ2>;
    ...
```

Die Bezeichner sind damit als Typbezeichner definiert.
Diese kann man dann in einer Variablen-Deklaration benutzen:

```
VAR
    <Bezeichner-Liste1>: <Bezeichner1>;
    <Bezeichner-Liste2>: <Bezeichner2>;
    ...
```

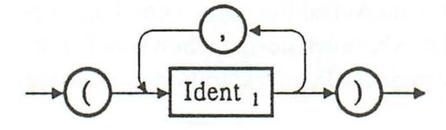
Die nächsten Abschnitte verschaffen über diese Verwendung Klarheit.

1.6.1 Aufzählungstypen

Des öfteren hat man beim Programmieren Daten, die in Form einer Aufzählung vorkommen, also nur mehrere feste Werte annehmen. Beispiele sind Wochentage, Monate, Spielkarten, Geschlecht usw. In primitiveren Programmiersprachen wie Basic verwendet man zu ihrer Darstellung ganze Zahlen. Das hat den Nachteil, daß man – nehmen wir einmal das Beispiel Spielkarten – jeder Spielkarte eine bestimmte Nummer zuordnen muß. Beim Programmieren muß man diese Zuordnung (zum Beispiel »As« = 8) dauernd parat haben. Ein Vorteil dieser Numerierung ist aber, daß man damit FOR-Schleifen bilden und Kartenwerte erhöhen bzw. erniedrigen kann, oder sie als Selektoren einer CASE-Anweisung nutzen kann.

Modula-2 bietet hier die »Aufzählungstypen«, die diese Vorteile ebenfalls besitzen, ohne daß man beim Programmieren die Zuordnung zu einer Nummer im Kopf haben muß. Das macht der Compiler.

Für Aufzählungstypen deklariert man eine Folge von Bezeichnern, die man durch Kommata trennt und in runden Klammern einschließt:



SYNTAX: "Enumeration"(15)

Beispiele:

```
TYPE
    Wochentag = (Montag, Dienstag, Mittwoch, Donnerstag,
                 Freitag, Samstag, Sonntag);
    Monat = (Jan, Feb, Mae, Apr, Mai, Jun,
             Jul, Aug, Sep, Okt, Nov, Dez);
    Karte = (Sieben, Acht, Neun, Zehn, Bube, Dame, Koenig, As);
```

Eine Variable `Tag` läßt sich so vom Typ `Wochentag` deklarieren:

```
VAR Tag: Wochentag;
```

und kann dann genau die Werte `Montag` bis `Sonntag` annehmen.

Den Standardtyp `BOOLEAN` kann man sich ebenfalls als Aufzählungstyp denken:

```
TYPE BOOLEAN = (FALSE, TRUE);
```

Intern wird ein Aufzählungstyp ähnlich einer CARDINAL-Zahl als Dualzahl dargestellt und belegt (je nach Compiler oder Größe) ein oder zwei Byte. Das erste Element eines Aufzählungstypen wird wie die CARDINAL-Zahl `0` dargestellt, das zweite wie eine `1` usw.

Beim Beispiel `Wochentag` wird also `Montag` wie `0` und `Sonntag` als duale `6` abgespeichert. Das impliziert gleichzeitig, daß

```
Montag < Dienstag < Mittwoch <...< Sonntag
```

ist. Der Ausdruck

`Montag < Donnerstag` liefert also `TRUE`.

Ebenso sind folgende Konstruktionen möglich:

```
IF Tag = Montag THEN WriteString("Wochenende vorbei") END;
IF Tag <= Freitag THEN arbeiten ELSE feiern END;
```

Aufzählungstypen gehören zu den skalaren Typen. Deshalb kann man sie auch zur Steuerung von `FOR`-Schleifen einsetzen:

```
VAR Tag: Wochentag;
...
FOR Tag := Montag TO Sonntag DO <Anweisungen> END;
```

oder, falls man die Reihenfolge der Wochentage nicht im Kopf hat, gleichbedeutend:

```
FOR Tag :=MIN(Wochentag) TO MAX(Wochentag) DO <...> END;
```

Aufzählungstypen sind auch praktisch für CASE-Anweisungen:

```
VAR m: Monat;
...
CASE m OF
  Januar..April, September..Dezember:
    WriteString("Ein Monat mit 'r', also Muscheln essen!")
  ELSE
    WriteString("Besser keine Muscheln essen!")
END;
```

Die Standardprozeduren `INC` und `DEC` lassen sich ebenfalls verwenden. Nach

```
Tag := Montag;
INC(Tag,3)
```

enthält `Tag` den Wert `Donnerstag`.

Achtung:

Von `Montag` gibt es keinen Vorgänger. Genauso wenig gibt es zu `Sonntag` einen Nachfolger. Also ist nach

```
Tag := Montag;
DEC(Tag);
```

die Variable `Tag` nicht definiert und liefert eventuell einen Laufzeitfehler.

Notfalls kann man Aufzählungstypen auch auf `CARDINAL`-Werte abbilden. Dazu gibt es die Funktionen `ORD` und `VAL`. Erstere wandelt den Wert eines Aufzählungstypen in seine entsprechende `CARDINAL`-Zahl, `VAL` macht das Gegenteil.

Wichtig:

`VAL` muß dazu aus dem Modul `SYSTEM` importiert werden. Dazu braucht man in der Importliste `FROM SYSTEM IMPORT VAL;`
`VAL` hat noch eine Besonderheit: es hat als ersten Parameter einen Typ!
`VAL` muß schließlich wissen, in welchen Typ es die `CARDINAL` -Zahl verwandeln soll.

So liefert

`ORD(Mittwoch)` die `CARDINAL`-Zahl `2` , umgekehrt liefert
`VAL(Wochentag, 2)` den Wert `Mittwoch`.

Für Werte von Aufzählungstypen gibt es keine Ein- und Ausgabeprozeduren. So etwas wie `WriteString(Montag)` kann nicht funktionieren, schließlich ist `Montag` kein String, sondern ein Bezeichner!

Aus diesen Beispielen sollte hervorgegangen sein, wie einfach Aufzählungstypen zu handhaben sind. Wir werden sie im folgenden öfter benutzen, da sie die Lesbarkeit der Programme erhöhen.

1.6.2 Unterbereichstypen

Eine weitere Klasse von Datentypen, die der Benutzer selbst definieren kann, sind die »Unterbereichstypen«. Ein Unterbereichstyp bildet einen zusammenhängenden Bereich eines anderen skalaren Typs – seines Basistyps – und stellt somit wieder einen skalaren Typ dar.

Die Deklaration sieht im einfachsten Fall so aus:

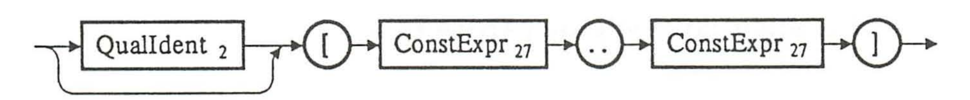
```
TYPE <NeuerTyp> = [<Untergrenze>..<Obergrenze>];
```

Beispiele:

```
    TYPE
        Kleinbuchstaben = ["a".."z"];
        Ziffern         = ["0".."9"];
        Zweistellig     = [10..99];
        Arbeitstag      = [Montag..Freitag];
        Wochenende      = [Samstag..Sonntag];
```

Die beiden ersten Typen `Kleinbuchstaben` und `Ziffern` stellen Unterbereichstypen des Datentyp `CHAR` dar. Die Typen `Arbeitstag` und `Wochenende` sind Unterbereiche des Basistyps `Wochentag` aus dem vorigen Abschnitt. Der dritte Typ `Zweistellig` ist ein Unterbereich vom Basistyp `CARDINAL`, da der Compiler die Konstanten `10` und `99` als `CARDINAL`-Konstanten ansieht. Nun hätte man `Zweistellig` aber auch als Unterbereich des Typs `INTEGER` haben wollen, schließlich sind `10` und `99` auch gültige `INTEGER`-Konstanten. Deshalb ist es auch erlaubt, bei der Definition zusätzlich den Basistypen mit anzugeben:

```
TYPE <NeuerTyp> = <BasisTyp>[<Untergrenze>..<Obergrenze>];
```



SYNTAX: "Subrange"(16)

Beispiele:

```
    Kleinbuchstaben       = CHAR["a".."z"]
    ZweistelligInteger    = INTEGER [10..99];
    ZweistelligCardinal   = CARDINAL[10..99];
```

Ober- und Untergrenze dürfen auch einen konstanten Ausdruck darstellen:

```
    CONST mitte = 100;
    TYPE  Bereich = [mitte - 20 .. mitte + 20];
```

Es sei noch einmal betont, daß ein Unterbereichstyp stets einen zusammenhängenden Teil (aufsteigende Folge ohne Lücke, unterer Grenzwert oberer Grenzwert) darstellt. Die Deklaration

```
    TYPE Vokale = ["A","E","I","O","U"]; (*geht nicht! *)
```

ist falsch! Auch gibt es keine REAL-Unterbereichstypen, da REAL kein aufzählbarer (skalarer!) Typ ist:

```
TYPE Intervall = [0.0..1.0]; (*geht nicht! *)
```

Wozu braucht man Unterbereichstypen?

Wie bei Aufzählungstypen gibt es wiederum zwei Gründe:

a) Erhöhung der Lesbarkeit
b) Erhöhung der Sicherheit

Zu a)
Durch die Einschränkung auf den betreffenden Unterbereich wird dem Leser des Programmes klar, daß Variablen dieses Unterbereichtyps nur die Werte innerhalb der vorgegebenen Grenzen annehmen werden.

Zu b)
Zur Laufzeit kann überprüft werden, ob die Variablen eines Unterbereichs nur gültige Werte annehmen. Wenn dann das Programm wegen einer unzulässigen Zuweisung abbricht, sollte das dem Programmierer Anlaß zum Nachdenken geben.

1.6.3 Die Datenstruktur »Feld« (ARRAY)

Wir kommen nun zum ersten »strukturierten« Datentyp, dem Feld oder ARRAY (engl. *array* = »Feld«). Schauen wir uns das Programm milderLehrer aus dem Abschnitt 1.4.2 an.

Hier benutzten wir die Variablen note1, note2, ... , note6 um die Anzahl der erteilten Noten »1« bis »6« abzuspeichern. Für solche Variablen, die vom gleichem Typ sind und mehrfach auftreten verwendet man Felder. In unserem Beispiel benötigen wir sechs Variablen für die Anzahl der Noten »1« bis »6«:

```
VAR note: ARRAY [1..6] OF CARDINAL;
```

[1..6] ist hier der sogenannte Indextyp, CARDINAL der Basistyp. Damit sind nun die sechs Variablen note[1], note[2], note[3], note[4], note[5], note[6] definiert. Auf die i-te Note greift man also mit note[i] zu. »i« ist dann ein »Index« des Feldes note und note[i] ist ein »Element« des Feldes.

Felder lassen sich sehr gut mit FOR-Schleifen bearbeiten. So setzt die nächste Anweisung einfach alle Notenzahlen auf Null:

```
FOR i := 1 TO 6 DO note[i] := 0 END;
```

Wenn wir auch noch die Würfel in einem Feld

```
TYPE wuerfel := ARRAY[1..3] OF CARDINAL;
```

abspeichern, läßt sich das Programm rasch von 42 auf 28 Zeilen verkürzen.

```
MODULE milderLehrerMitArray;

FROM RandomGen IMPORT Randomize, RandomCard;
FROM InOut     IMPORT WriteCard, WriteReal, WriteLn, WriteString, KeyPressed;

VAR wuerfel             : ARRAY [1..3] OF CARDINAL;
    note                : ARRAY [1..6] OF CARDINAL;
    i, anzahl, minimum  : CARDINAL;

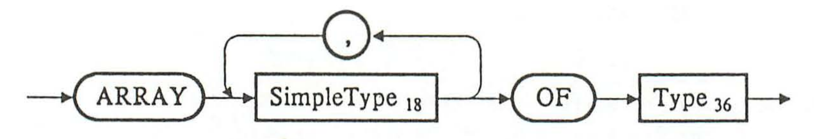
BEGIN
  WriteString("Der milde Lehrer würfelt und würfelt, ");
  WriteString("bis Sie eine Taste drücken..."); WriteLn; WriteLn;
  anzahl:=0;
  FOR i := 1 TO 6 DO note[i] := 0 END;
  Randomize(12345);
  REPEAT
    FOR i := 1 TO 3 DO wuerfel[i] := RandomCard(1,6) END;
    minimum := wuerfel[1];
    FOR i := 2 TO 3 DO
      IF minimum > wuerfel[i] THEN minimum := wuerfel[i] END
    END;
    INC(note[minimum]); INC(anzahl);
  UNTIL KeyPressed();
  WriteString("Anzahl der erwürfelten Noten insgesamt: "); WriteCard(anzahl,10);
  WriteLn; WriteLn;
  FOR i := 1 TO 6 DO
    WriteReal(FLOAT(note[i]) * 100.0 / FLOAT(anzahl), 6, 2);
    WriteString("% für Note"); WriteCard(i,2); WriteLn;
  END;
  REPEAT UNTIL KeyPressed()
END milderLehrerMitArray.
```

Mit einem Element eines Feldes kann man also rechnen wie mit einer Variablen des Basistyps. Auf Feldern selbst sind keine Operatoren definiert. Will man aber ein Feld in ein anderes kopieren, so braucht man, sofern beide Felder vom gleichem Typ sind, nicht jedes Element einzeln zu kopieren. Statt

```
TYPE feld = ARRAY [1..99] OF CARDINAL;
VAR  a,b: feld;
     i:   CARDINAL;
...
FOR i := 1 TO 99 DO a[i] := b[i] END;
```

kann man direkt `a:=b;` schreiben, wohlgemerkt nur, wenn beide Felder vom gleichen Typ sind.

Allgemein sieht die Deklaration für ein Feld wie folgt aus:



SYNTAX: "ArrayType"(19)

SimpleType beschreibt einen Indexdatentyp. Jeder Indextyp muß dabei ein Aufzählungstyp, ein Unterbereichstyp oder einer der Standardtypen `CHAR` und `BOOLEAN` sein. *Type* beschreibt den Basistyp des Feldes, also den Datentyp der einzelnen Elemente des Feldes. Im vorstehendem Beispiel also `CARDINAL`. Folgendes ist also möglich:

```
TYPE
    Monat           = (Jan, Feb, Mae, Apr, Mai, Jun,
                       Jul, Aug, Sep, Okt, Nov, Dez);
    Tagzahl         = [28..31];
    Geschlecht      = (m,w);
    Kleinbuchstaben = ["a".."z"];

VAR
    TageImMonat : ARRAY Monat OF Tagzahl;
    Bewohner    : ARRAY Geschlecht OF CARDINAL;
    Haeufigkeit : ARRAY Kleinbuchstaben OF REAL;
    Zeichenfeld : ARRAY CHAR OF REAL;
```

Falsch wären die Typen:

```
    ARRAY REAL OF INTEGER;
    ARRAY BITSET OF REAL;
```

da weder `BITSET` noch `REAL` zulässige Indextypen sind.

Mehrdimensionale Felder

Aus dem Syntaxdiagramm ersieht man , daß auch folgendes möglich ist:

```
TYPE Farbe = (schwarz,weiss);
VAR Schachbrett: ARRAY [1..8],["A".."H"] OF Farbe;
```

Wir haben hier ein sogenanntes zweidimensionales Feld. Es verkürzt praktisch die Deklaration:

```
VAR Schachbrett: ARRAY [1..8] OF ARRAY ["A".."H"] OF Farbe;
```

Ein zweites Beispiel:

```
TYPE matrix = [1..3],[1..2] OF REAL;
VAR  m: matrix;
```

Hier wird eine Matrix aus `3*2` Zahlen gebildet. Der Zugriff auf eine einzelne Komponente kann mit `m[i][j]` oder einfacher mit `m[i,j]` erfolgen. Stellen Sie sich unter diesem zweidimensionalen Feld eine Tabelle bestehend aus 6 Zahlen in 3 Spalten und 2 Zeilen vor.

Ein Programmbeispiel mit zweidimensionalen Feldern: Bekanntlich gilt für die Binomialkoeffizienten des Pascal'schen Dreiecks

$$\binom{n}{0} = 1, \quad \binom{n}{n} = 1 \text{ und } \quad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{für } n>0, k>0$$

Die Koeffizienten können mit dieser Rekursionsformel als zweidimensionales Feld errechnet werden:

```
MODULE PascalDreieck;

FROM InOut IMPORT  Write, WriteString, WriteCard, WriteLn, Read;

CONST max = 11;

VAR
    BinKoeff : ARRAY[0..max],[0..max] OF CARDINAL;
    n,k,l    : CARDINAL;
    taste    : CHAR;

PROCEDURE KoeffizientenBerechnen;
BEGIN
  FOR n:=0 TO max DO
```

```
      BinKoeff[n,0] := 1; BinKoeff[n,n] := 1
    END;
    FOR n:=1 TO max DO
      FOR k:=1 TO n-1 DO
        BinKoeff[n,k] := BinKoeff[n-1,k-1] + BinKoeff[n-1,k]
      END
    END
  END KoeffizientenBerechnen;

  BEGIN
    WriteString(" Pascalsches Dreieck");
    KoeffizientenBerechnen;
    FOR n := 0 TO max DO
      WriteLn; FOR k := 1 TO 33-3*n DO Write(" ") END;
      FOR k := 0 TO n DO WriteCard(BinKoeff[n,k],6) END;
    END;
    Read(taste)
  END PascalDreieck.
```

In analoger Weise lassen sich mehrdimensionale Matrizen bilden.

Internes zu Feldern

Nehmen wir an, wir haben ein Feld

`VAR feld: ARRAY [5..9] OF LONGCARD;`

Eine Adresse im Computer ist nichts anderes als die Nummer eines Speicherplatzes. Bei unserem Feld stimmt die Basisadresse des Feldes (also der Speicherplatz, an dem das Feld beginnt) mit der Adresse von `feld[5]` überein. Wenn der Computer zum Beispiel auf `feld[8]` zugreifen soll, geht er folgendermaßen vor, um die Adresse von `feld[8]` zu ermitteln:

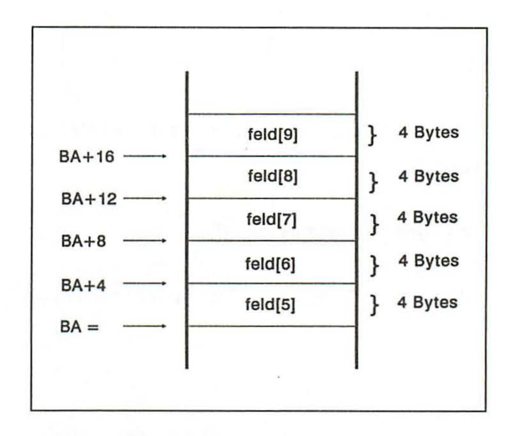
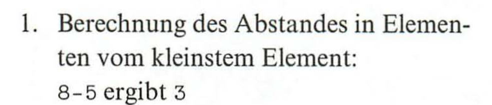


Bild 1.18: Speicherung des Feldes VAR feld : ARRAY [5..9] OF LONGCARD



1. Berechnung des Abstandes in Elementen vom kleinstem Element:
   `8-5` ergibt `3`
2. Er multipliziert diesen Abstand mit der Speichergröße eines Elementes (hier 4 Byte für LONGCARD):

`3*4 Byte` ergibt `12 Byte.`
Damit erhält er den Abstand in Byte.

3. Er addiert diesen Wert zur Basisadresse des Feldes. Damit erhält er die Adresse des Elementes:
   `Adresse = Basisadresse + Byteabstand`

Nun, das alles benötigt Zeit und Code. In geschwindigkeitskritischen Fällen kann der ersten Schritt vermieden werden, indem man als kleinsten Index 0 vorsieht, in unserem Beispiel also

```
VAR feld: ARRAY [0..4] OF LONGCARD;
```

deklariert.

Aus der Darstellung ist ersichtlich, daß sich der Speicherbedarf einer Feldes errechnet zu

```
Anzahl der Elemente * Speicherbedarf des Basistyps
```

Das Feld

```
VAR DreiDimensionen: ARRAY [0..9],[0..9],[0..9] OF LONGCARD;
```

belegt also `10*10*10*4 Byte.`

Der Speicher als Feld

Es wird klar, daß man den Speicher selbst als Feld von Bytes auffassen kann. Daher ist es auch möglich, einer bestimmten Variablen einen festen Speicherplatz zuzuordnen. Dies kann unter anderem sinnvoll sein, wenn man bestimmte Speicherstellen lesen oder verändern will, in denen das Betriebssystem gewisse Informationen abgelegt hat. Ein Beispiel könnten die Systemvariablen des Atari sein. Wir bringen ein Programm dazu im Abschnitt 3.3. Soll die `CARDINAL`-Variable `c` an der Adresse »FFE« (Hexadezimal) stehen, deklariert man in Modula

```
VAR c[0FFEH]: CARDINAL;
```

Open-ARRAY-Parameter

Schauen wir uns abschließend die Bestimmung des Minimums der Würfel im Programm »`MilderLehrerMitArray`« an. Man könnte daraus eine schöne Prozedur schreiben.

Gegeben sei der Typ

```
TYPE
    bereich = [5..10];
    feldtyp = ARRAY bereich OF CARDINAL;
```

und die Prozedur

```
PROCEDURE minimum(feld: feldtyp): CARDINAL;

VAR
    i    : bereich;
    mini : CARDINAL;

BEGIN
  mini := feld[MIN(bereich)];
  FOR i := MIN(bereich) + 1 TO MAX(bereich) DO
    IF feld[i] < mini THEN mini := feld[i] END;
  END;
  RETURN mini
END minimum;
```

Es stört hier ein wenig, daß man mit dieser Prozedur nicht beliebige `CARDINAL`-Felder nach dem Minimum durchsuchen kann, sondern nur `CARDINAL`-Felder vom Typ `feldtyp`, also mit dem Indexbereich `[5..10]`, denn schließlich muß ja der Typ in der Parameterliste angegeben werden. Schöner wäre es, eine allgemein nützliche Prozedur zu definieren, die für jedes `CARDINAL`-Feld funktioniert, unabhängig von dem aktuellen Indexbereich. Modula kennt hier den »Open-Array«-Typ (*open array* = »offenes Feld«), einen »JokerFeldtyp«, bei dem innerhalb einer Parameterliste der Indexbereich nicht angegeben werden muß!

Wir können also im Prozedurkopf schreiben:

```
PROCEDURE minimum(feld:ARRAY OF CARDINAL): CARDINAL;
```

Für die Steuerung der `FOR`-Schleife benötigen wir aber die Indexgrenzen. Wie kommen wir jetzt daran? Der kleinste Index wird bei einem offenen Feld stets als `0` (Null) angenommen; den größten Index erhält man mit `HIGH(feld)`. Die Indexgrenzen werden also beim Übergeben »transponiert«. Beispielsweise wird ein Feld, das als `ARRAY[20..25]` deklariert ist, als `ARRAY[0..5]` übergeben.

Einem Open-Array-Parameter kann man nur ein Feld beliebiger Größe des gleichen Basistyps übergeben. Das folgende Demonstrationsprogramm macht dies klar:

```
MODULE OffenesFeldDemo;

FROM InOut IMPORT Read, WriteCard, WriteString, WriteLn;

PROCEDURE minimum(feld : ARRAY OF CARDINAL) : CARDINAL;   (* 'offenes' Feld *)
VAR min, i :CARDINAL;
```

```
BEGIN
  min := feld[0];
  FOR i := 1 TO HIGH(feld) DO
    IF min > feld[i] THEN min := feld[i] END
  END;
  RETURN min
END minimum;

VAR
    feld1 : ARRAY [5..8] OF CARDINAL;
    feld2 : ARRAY ["A".."D"] OF CARDINAL;
    feld3 : ARRAY BOOLEAN OF CARDINAL;
    feld4 : ARRAY (rot, gelb, blau) OF CARDINAL;
    feld5 : ARRAY [6..6] OF CARDINAL;          (* Feld mit nur einem Element *)
    taste : CHAR;
BEGIN
  feld1[5] := 5; feld1[6] := 7; feld1[7] := 2; feld1[8] := 10;
  feld2["A"] := 65; feld2["B"] := 66; feld2["C"] := 67; feld2["D"] := 68;
  feld3[FALSE] := 100; feld3[TRUE] := 50;
  feld4[rot] := 30; feld4[gelb] := 20; feld4[blau] :=10;
  feld5[6] := 3;
  WriteString("Minimum von feld1: "); WriteCard(minimum(feld1),2); WriteLn;
  WriteString("Minimum von feld2: "); WriteCard(minimum(feld2),2); WriteLn;
  WriteString("Minimum von feld3: "); WriteCard(minimum(feld3),2); WriteLn;
  WriteString("Minimum von feld4: "); WriteCard(minimum(feld4),2); WriteLn;
  WriteString("Minimum von feld5: "); WriteCard(minimum(feld5),2); WriteLn;
  Read(taste)
END OffenesFeldDemo.
```

Man sieht also, daß der Open-Array-Parameter eine sehr flexible Programmierung erlaubt. Nur mehrdimensionale Arrays lassen sich nicht als Open-Array-Parameter übergeben.

Es stellt sich die Frage, wie das funktioniert. Woher »kennt« die Prozedur `HIGH` die obere Indexgrenze?

Bei Open-Array-Parametern wird neben der Basisadresse des Feldes noch der `HIGH`-Wert (die Anzahl der Elemente minus 1) wie ein zusätzlicher Parameter übergeben. Das ist das ganze Geheimnis.

Im Kapitel 2.1 gehen wir tiefer auf die Datenstruktur Feld ein. Erklärtes Ziel dieses Buches ist es dabei, möglichst flexible Prozeduren zu schreiben. Die Prozedur `minimum` mit einem offenen Feldparameter wäre ein erstes einfaches Beispiel hierzu.

Strings als Felder

Zeichenketten oder Strings werden in Modula als `ARRAY [0..max] OF CHAR` deklariert. Eine Variable dieses Typs kann höchstens max + 1 Zeichen aufnehmen. Falls nicht alle Zeichen belegt sind, wird als Endemarkierung hinter dem letzten gültige Zeichen `0C` gesetzt. Die Länge eines Strings läßt sich demnach mit folgender Prozedur bestimmen:

```
PROCEDURE StrLaenge(VAR s :ARRAY OF CHAR) : CARDINAL;
VAR  i : CARDINAL
BEGIN
  i=0;
  WHILE (s[i] # 0C) AND (i<=HIGH(s)) DO INC(i) END
  RETURN i
END StrLaenge;
```

Solche Prozeduren zur Stringbehandlung findet man bereits fertig im Modul `Strings`, der zu jedem Modula-System mitgeliefert wird.

Es sei nun

```
VAR    s : ARRAY [0..9] OF CHAR ;
```

statt nun

```
s[0]:="A"; s[1]:="T"; s[2]:="A"; s[3]:="R"; s[4]:="I"; s[5]:=0C;
```

zu setzen, gibt es speziell für Zeichenketten die wesentlich einfachere Möglichkeit, einer Variablen einen konstanten String zuzuweisen:

```
s:="ATARI";
```

leistet dasselbe. Es wird hierbei automatisiert `s[5]:=0C` gesetzt. Die weiteren Zeichen sind dann zufällig belegt.

1.6.4 Die Datenstruktur »Verbund« (RECORD)

Es gibt Dinge, die gehören einfach zusammen! Mit der Datenstruktur Feld lassen sich ausschließlich Elemente gleichen Typs behandeln. Oft sollen jedoch Daten verschiedenen Typs verarbeitet werden, die man unter einem gemeinsamen Oberbegriff zusammenfassen möchte.

Beispiel hierfür wäre eine Anschrift, bestehend aus Straße, Hausnummer, Postleitzahl (PLZ) und Wohnort. In Modula drückt man diesen »Verbund« so aus:

```
TYPE String = ARRAY[0..29] OF CHAR;
     Adresse  = RECORD
                   Strasse : String;
                   HausNr  : CARDINAL;
                   Plz     : [1000..9999];
                   Wohnort : String
                END;
```

Haben wir die Variablen

```
VAR
    Anschrift1, Anschrift2: Adresse;
```

so bezeichnet `Anschrift1.Strasse` einen String. Mit `Anschrift1` wird also auf den gesamten Verbund zugegriffen, mit `Anschrift1.`*<Bezeichner>* auf eine einzelne »Komponente« (in der Literatur auch mit »Feld« bezeichnet; wir unterlassen das, da eine Begriffsverwechselung mit `ARRAY` entstehen könnte).

Folgende Zuweisungen sind also korrekt:

```
Anschrift1.Strasse := "Hohe Gasse";
Anschrift1.HausNr := 315;
Anschrift1.Plz := 5600;
Anschrift1.Wohnort := "Wuppertal 1";
Anschrift1.Strasse[1] := "ö";
Anschrift2 := Anschrift1;
```

Aus dem letztem Beispiel ist ersichtlich, daß man Verbunde gleichen Typs wie Felder einander zuweisen kann.

Faßt man die Unterschiede von Feldern und Verbunden zusammen, so ergibt sich:

- Felder sind Zusammenfassungen von Daten gleichen Typs, Verbunde sind Zusammenfassungen von Daten verschiedenen Typs.
- Auf eine einzelne Feldkomponente kann man mit einem Ausdruck zugreifen (z.B. `feld[5*3+i]`), bei einem Record wird die Komponente durch ihren Bezeichner mit einem vorangestellten Punkt selektiert (`Anschrift1.Strasse`).

Internes Datenformat

Die Abspeicherung eines Verbundes erfolgt komponentenweise nacheinander. Also errechnet sich der gesamte Speicherplatzbedarf eines Verbundes als Summe seiner Komponenten. Der Datentyp `Adresse` belegt demnach

```
   30 Byte (30*1 Byte für CHAR)
+   2 Byte (CARDINAL)
+   2 Byte (Unterbereichstyp von CARDINAL)
+  30 Byte (30*1 Byte für CHAR)
---------
=  64 Byte für den Datentyp Adresse
```

Anmerkung:

In der Praxis kann es vorkommen, daß der Verbund etwas mehr Speicherplatz belegt als die Summe seiner Komponenten. Das liegt daran, daß aus technischen Gründen »Füllbytes« eingeschoben werden müssen, wenn eine Komponente eine ungerade Zahl von Bytes belegt. Der 68000-Prozessor liebt nun mal gerade Adressen.

Die WITH-Anweisung

Im obigen Beispiel weisen wir zu:

```
Anschriftl.Strasse := ...;
Anschriftl.HausNr := ...;
Anschriftl.Plz := ...;
Anschriftl.Wohnort := ...;
```

Diese sperrige Konstruktion läßt sich mit der `WITH`-Anweisung vereinfachen zu

```
WITH Anschriftl DO
    Strasse := ...;
    HausNr := ...;
    Plz := ...;
    Wohnort := ...
END;
```

Innerhalb der `WITH`-Anweisung sind `Strasse`, `HausNr`, `Plz` und `Wohnort` bekannt. Die `WITH`-Anweisung spart also Schreibarbeit. Außerdem kann sie effektiver sein, wenn in der `WITH`-Klausel (der Ausdruck zwischen `WITH` und `DO`) eine indizierte Feldvariable ist, da hier der Index nur einmal ausgerechnet zu werden braucht. `WITH`-Anweisungen dürfen natürlich auch geschachtelt werden.

Ein komplexes Beispiel

Selbstverständlich können die Komponenten wiederum Verbunde (oder auch Felder) sein. Die folgende Deklaration zeigt ein komplexes Beispiel:

```
CONST
    maxZeichen = 29;
    maxBelegung = 1000;

TYPE String = ARRAY [0..maxZeichen] OF CHAR;
     Datum = RECORD
                Tag   : [1..31];
                Monat : [1..12];
                Jahr  : CARDINAL;
              END;
    Adresse = RECORD
                   Strasse : String;
                   HausNr  : CARDINAL;
                   Plz     : [1000..9999];
                   Wohnort : String
                END;
    NameTyp = RECORD Vorname, Nachname: String END;
    Mitarbeiter = RECORD
                    Name          : NameTyp
                    Anschrift     : Adresse;
                    Geburtstag    : Datum;
                    Konfession    : (ev,rk,so);
                    Gehalt        : Real
                  END;

VAR Belegschaft: ARRAY [0..maxBelegung] OF Mitarbeiter;
```

Der hierarchische Aufbau des Feldes `Belegschaft` läßt sich in einem Datenstrukturbaum graphisch darstellen:

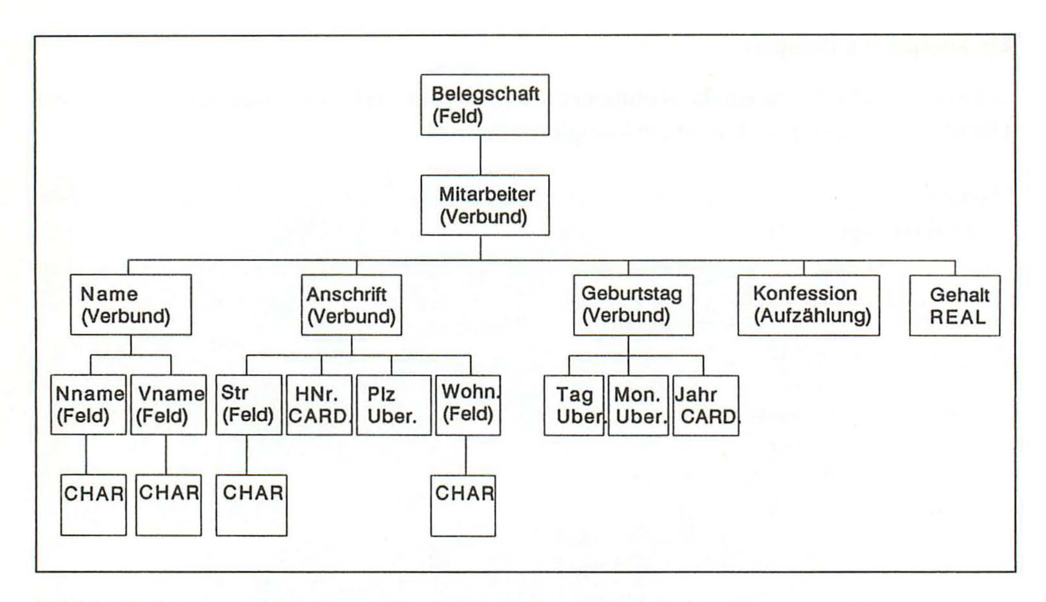


Bild 1.19: Datenstruktur-Baum für die Variable >Belegschaft<

An den »Blättern« des Baumen, also dort unten, wo es nicht mehr weitergeht, sind die Datentypen elementar. Es bezeichnet

`Belegschaft`:
: Die gesamte Belegschaft eines Betriebes
: Typ: `ARRAY [0..maxBelegung] OF Mitarbeiter`

`Belegschaft [5]`:
: Den 5. Mitarbeiter
: Typ: `Mitarbeiter` (Verbund)

`Belegschaft[5].Name`:
: seinen Namen
: Typ: `NameTyp` (Verbund)

`Belegschaft[5].Name.Nachname`:
: seinen Nachnamen
: Typ: `String` (= `ARRAY [0..29] OF CHAR`)

`Belegschaft[5].Name.Nachname[0]`:
: Den ersten Buchstaben seines Nachnamens
: Typ: `CHAR`

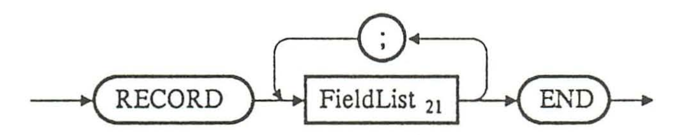
Ein Geburtsdatum weist man dem siebten Mitarbeiter wie folgt zu:

```
WITH Belegschaft[7].Geburtstag DO
  Tag := 24; Monat := 12; Jahr := 0
END;
```

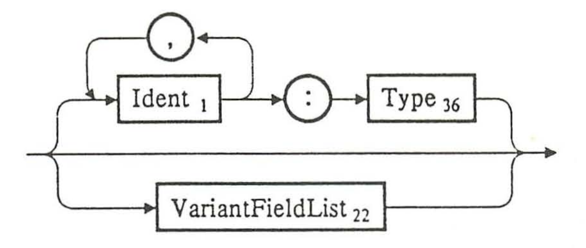
Man erkennt an diesen Beispielen, daß die Datenstruktur Verbund fundamental für Datenverwaltungsaufgaben ist. Daher werden wir im Abschnitt über Dateien 2.3 darauf zurückkommen. Darüber hinaus gibt es in Modula Möglichkeiten, komplexe Datenstrukturen und »Abstrakte Datentypen« zu formulieren; auch hierauf gehen wir im gesamten Kapitel 2 ein.

Die Syntax von Verbunden und varianten Verbunden

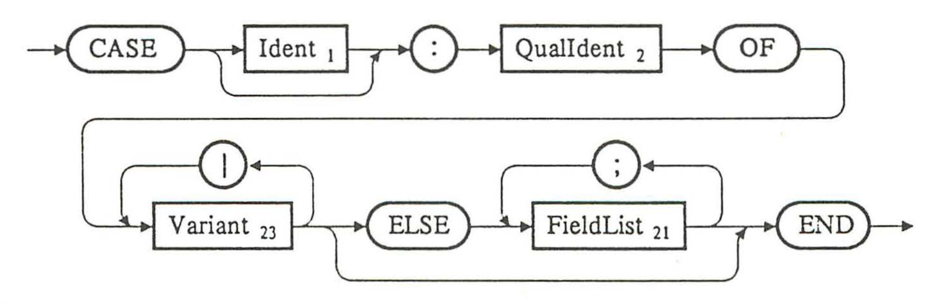
Vergleichen Sie die folgenden Syntaxdiagramme mit unseren Beispielen!



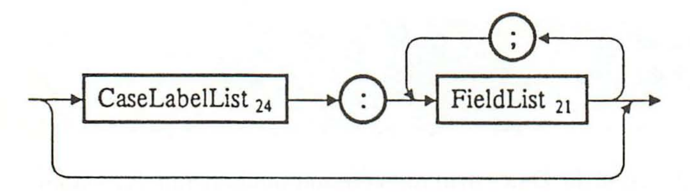
SYNTAX: "RecordTyp"(20)



SYNTAX: "FieldList"(21)



SYNTAX: "VariantFieldList"(22)



SYNTAX: "Variant"(23)

Es dürfte auffallen, daß es da noch einen längeren CASE-Zweig gibt, den wir noch nicht besprochen haben. Dies ist wirklich eine interessante Angelegenheit, denn damit ist es möglich, zwei unterschiedlichen Variablen den gleichen Platz im Speicher einzuräumen! Diese Struktur heißt Varianter Verbund«. Beispiel:

```
MODULE VarianterRecordDemo;

FROM InOut IMPORT WriteCard, WriteInt, WriteString, WriteLn, Read;

VAR taste: CHAR;
    VarRec: RECORD
              CASE Selektor: BOOLEAN OF
                FALSE: i:INTEGER |
                TRUE:  c:CARDINAL
              END
            END;

BEGIN
  VarRec.Selektor := FALSE; VarRec.i := -1;
  WriteLn; WriteString("Wert als INTEGER:  ");
  VarRec.Selektor := FALSE; WriteInt(VarRec.i,1);
  WriteLn; WriteString("Wert als CARDINAL: ");
  VarRec.Selektor := TRUE; WriteCard(VarRec.c,1);
  Read(taste)
END VarianterRecordDemo.
```

Das Programm gibt aus:

```
Wert als INTEGER:  -1
Wert als CARDINAL: 65535
```

Klar, wir belegen in der ersten Programmzeile `VarRec.i` mit -1. Intern:

```
11111111 11111111 (Dual)
```

Nun hat `VarRec.c` aber denselben Speicherplatz. Also wird bei `VarRec.c` diese Zahl als Kardinalzahl interpretiert. Die `CARDINAL`-Zahl mit diesem Bitmuster entspricht aber 65535.

Die Variable `Selektor` gibt im allgemeinen an, welche »Variante« des Speicherplatzes (hier: `i` oder `c`) gerade gültig ist. Man kann den Selektor auch weglassen, damit erspart man sich das Umschalten auf `TRUE` bzw. `FALSE` beim Zugriff auf die Varianten.

```
VAR VarRec: RECORD
              CASE : BOOLEAN OF
                FALSE: i: INTEGER |
                TRUE:  c:CARDINAL
              END;
            END;
```

In einem Record können natürlich variante und nichtvariante Teile vorkommen. Zum Beispiel:

```
Type AtomKern = RECORD
                  Name          : ARRAY[0..19] OF CHAR;
                  Symbol        : ARRAY [0..1] OF CHAR;
                  Protonenzahl,
                  Neutronenzahl : CARDINAL;
                  CASE radioaktiv : BOOLEAN OF
                    TRUE: Halbwertszeit: REAL
                  END
                END;

VAR Elemente: ARRAY [1..1000] OF Atom;
...
WITH Elemente[1] DO
  Name := "Wasserstoff";
  Symbol := "H";
  Protonenzahl := 1;
  Neutronenzahl := 0;
  radioaktiv := FALSE
END;
WITH Elemente[800] DO
  Name := "Uran 238"
  Symbol := "U";
```

```
  Protonenzahl := 92;
  Neutronenzahl := 146;
  radioaktiv := TRUE;
  Halbwertszeit := 4.67E9        (* Jahre *)
END;
```

1.6.5 Die Datenstruktur »Menge« (SET)

Wir haben bereits den Standard-Datentyp `BITSET` kennengelernt. Er beschreibt die Teilmengen der Menge {0,1,...,15}. Der Datentyp `SET` (engl. *set* = »Menge«) verallgemeinert dieses Konzept. Ein `SET`-Typ wird mit

```
SET OF <BasisTyp>
```

angegeben.

Der Basistyp ist dabei entweder ein Aufzählungstyp oder ein Unterbereichstyp der Form `[0..<Obergrenze>]`. Der Basistyp darf nur eine bestimmte Anzahl von Elementen umfassen, bei den Compilern für den Atari meist 16; Megamax-Modula gestattet hier 65 536 Elemente. Da aber der Compiler für jedes Element ein Bit benötigt, gilt für den benötigten Speicherplatz:

`SET OF [0..n]` belegt n+1 Bits (also (n+7) DIV 8 Byte,da auf ganze Byte aufgerundet wird).

Eine Menge mit 16 Elementen belegt 2 Byte, das ist standardmäßig das Maximum. Die von Pascal-Programmierern so beliebte Menge `SET OF CHAR` ist deshalb bei einigen Modula-Compilern nicht machbar. Megamax erlaubt hingegen:

`SET OF CHAR`	belegt 32 Byte (256 Bit)
`SET OF [0..65535]`	belegt 8 Kbyte = 8192 Byte
`SET OF BOOLEAN`	belegt 1 Byte (bei Megamax)

Mit Mengen kann man genauso operieren wie mit dem Typ `BITSET`. Hierzu das folgende Programm:

```
MODULE MengenDemo;

FROM InOut IMPORT Read, WriteCard, WriteString, Write, WriteLn;

TYPE
    Bereich      = [0..9];
    ZiffernMenge = SET OF Bereich;
```

```
PROCEDURE SchreibMenge(m : ZiffernMenge);
VAR i     : Bereich;
    komma : BOOLEAN;
BEGIN
  Write("{");
  komma := FALSE;
  FOR i:=0 TO 9 DO
    IF i IN m THEN
      IF komma THEN Write(",") END;
      WriteCard(i,1); komma := TRUE
    END;
  END;
  Write("}")
END SchreibMenge;

CONST m1 = ZiffernMenge{1,2,3,4,5,6};
      m2 = ZiffernMenge{4,5,6,7,8,9};

VAR taste : CHAR;

BEGIN
WriteString("Programm zur Demonstration von Mengen");       WriteLn; WriteLn;
WriteString("Menge m1 lautet      "); SchreibMenge(m1);     WriteLn;
WriteString("Menge m2 lautet      "); SchreibMenge(m2);     WriteLn;
WriteString("Durchschnitt m1*m2 = "); SchreibMenge(m1*m2);  WriteLn;
WriteString("Vereinigung m1+m2 =  "); SchreibMenge(m1+m2);  WriteLn;
WriteString("Differenz m1-m2 =    "); SchreibMenge(m1-m2);  WriteLn;
WriteString("Differenz m2-m1 =    "); SchreibMenge(m2-m1);  WriteLn;
WriteString("sym. Diff. m1/m2 =   "); SchreibMenge(m1/m2);  WriteLn;
Read(taste)
END MengenDemo.
```

Dies liefert folgende Ausgabe:

```
Programm zur Demonstration von Mengen
Menge m1 lautet      {1,2,3,4,5,6}
Menge m2 lautet      {4,5,6,7,8,9}
Durchschnitt m1*m2 = {4,5,6}
Vereinigung  m1+m2 = {1,2,3,4,5,6,7,8,9}
Differenz    m1-m2 = {1,2,3}
Differenz    m2-m1 = {7,8,9}
symm. Diff.  m1/m2 = {1,2,3,7,8,9}
```

Es gibt auch Mengenkonstanten. Dabei muß der Mengentyp mit angegeben werden:

<Mengentyp>{ *<Elementeliste>*}

Diese können dann so verwendet werden:

```
TYPE ZiffernMenge = SET OF [1..9];

CONST mk   = ZiffernMenge{1,2,6};
      leer = ZiffernMenge{};          (* Angabe einer leeren Menge *)
```

Auch wenn die Menüs, bei denen man die Anfangsbuchstaben der Menüpunkte eintippen muß, beim Atari etwas aus der Mode gekommen sind, wollen wir einfache Menütechnik zur Demonstration von Mengen zeigen. Die Handhabung von den Atari-typischen Pull-down-Menüs wird in Abschnitt 4.7 gezeigt. Wichtig am Beispiel-Programm ist die Prozedur `LiesZeichen`. Man kann hier solange ohne Bildschirmecho Zeichen eintippen, bis ein gültiges Zeichen (festgelegt durch `okMenge`) eingegeben wird. Damit auch auf Kleinbuchstaben in `Meldung` reagiert wird, wird die Funktion `CAP` benutzt, allerdings nur dann, wenn ausdrücklich Großbuchstaben erwünscht sind (also die OK-Menge nur aus Großbuchstaben besteht). Statt der Prozedur `Meldung` werden in einem realistischen Programm nun andere Prozeduren stehen.

```
MODULE MenueDemo;

FROM InOut IMPORT BusyRead, Write, WriteString, GotoXY;

TYPE ZeichenMenge = SET OF CHAR;(* Akzeptiert nicht jeder Modula-Compiler*)

PROCEDURE LiesZeichen(okMenge : ZeichenMenge) : CHAR;
VAR ch : CHAR;
BEGIN
  REPEAT
    REPEAT BusyRead(ch) UNTIL ch # 0C;        (* Einlesen ohne Bildschirmecho *)
    IF okMenge * ZeichenMenge{"a".."z"} = ZeichenMenge{} THEN ch:=CAP(ch) END;
    IF NOT (ch IN okMenge) THEN Write(7C) ELSE Write(ch) END;
  UNTIL ch IN okMenge;
  RETURN ch
END LiesZeichen;

PROCEDURE Meldung(art : CHAR);
BEGIN
  GotoXY(10,18);
  WriteString("Sie wählten den Menüpunkt ");
```

```
  CASE art OF
    "A" : WriteString("Arbeiten");
    "R" : WriteString("Rechnen ");
    "D" : WriteString("Drucken ");
    "E" : WriteString("Ende    ");
  END;
END Meldung;

PROCEDURE Menue;
VAR wahl : CHAR;
BEGIN
  GotoXY(10,10); WriteString("A rbeiten");
  GotoXY(10,11); WriteString("R echnen");
  GotoXY(10,12); WriteString("D rucken");
  GotoXY(10,13); WriteString("E nde");
  GotoXY(10,15); WriteString("Ihre Wahl: ");
  REPEAT
    GotoXY(21,15);
    wahl :=LiesZeichen(ZeichenMenge{"A","R","D","E"});
    Meldung(wahl);
  UNTIL wahl = "E"
END Menue;

BEGIN
  Menue
END MenueDemo.
```

Wir werden die Prozedur `LiesZeichen` noch einmal aufgreifen bei der Besprechung von Strings (Abschnitt 1.7.3). Dort wird eine »mengenfreie« Version geboten.

1.6.6 Die Datenstruktur »Zeiger« (POINTER)

Bei unserer Spracheinführung haben wir des öfteren die Frage beantwortet, wie gewisse Variablen intern dargestellt werden. Schauen wir uns nun einmal den gesamten Speicher des Rechners zur Laufzeit eines Programms an.

Der Speicher zur Programm-Laufzeit

Im großen und ganzen sieht das bei allen Compilern ähnlich aus:

Die Pfeile deuten jeweils an, in welcher Richtung die Speicherbereiche gefüllt werden.

Die Adressen *A1* und *A5* sind vom Computersystem vorgegeben. Der Speicherbedarf der globalen Variablen wird durch die Variablendeklaration eines Programmes festgelegt. Der Compiler reserviert also hierfür den entsprechenden Speicherplatz, die Adresse *A4* ist ihm also bekannt. Ebenso ist die Adresse *A2* durch den Compiler nach Beendigung des Übersetzungsvorganges gegeben.

Des weiteren wird beim Programm-Ablauf beim Eintritt in eine Prozedur (bei ihrer »Inkarnation«) Speicherplatz für die Rücksprungadressen, die Parameter und die lokalen Variablen benötigt, der nach Verlassen der Prozedur wieder freigegeben wird. Dafür wird ein bestimmter Speicherbereich vorgesehen: der Stack (engl. *stack* = »Stapel«). Es bleibt also noch ein großer freier Bereich, der sogenannte Heap (engl. *heap* = »Halde«).

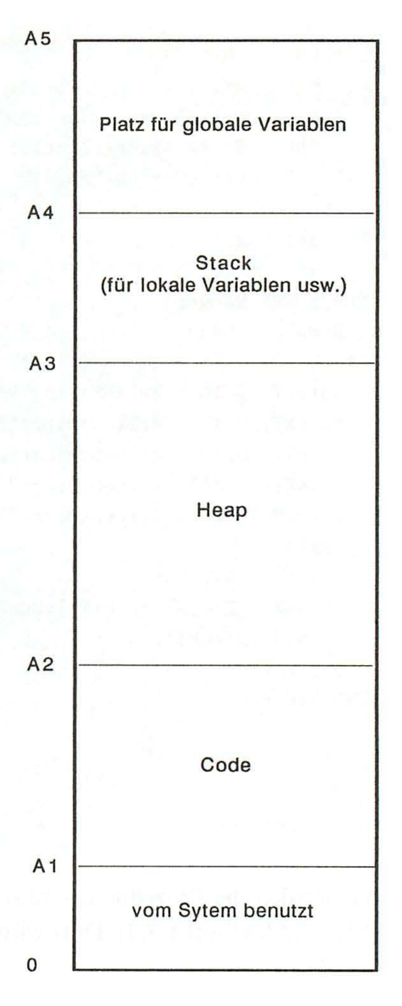


Bild 1.20: Speicheraufteilung zur Laufzeit eines Programms

In diesem Bereich kann der Modula-Programmierer auch Daten ablegen. Während aber die Verwaltung globaler und lokaler Variablen automatisch abläuft (das System »merkt« sich, wo es etwas hingeschrieben hat), muß sich der Programmierer bei Benutzung des Heap auch etwas mit der Speicherverwaltung auseinandersetzen. Außerdem muß er wissen, an welchem Speicherplatz (Adresse) welche Daten abgelegt wurden.

Dies hört sich kompliziert an, ist es aber nicht, weil Modula das sogenannte Pointer-Konzept (engl. *pointer* = »Zeiger«) unterstützt. Zeiger sind nichts anderes als Adressen, mit denen die Speicherplätze für unsere Variablen erreicht werden können. Zum Ablegen und Löschen stellt Modula die Prozedur `ALLOCATE` und `DEALLOCATE` aus dem Standardmodul `Storage` zu Verfügung. Doch bevor wir auf Einzelheiten gehen, sei die Frage erlaubt:

Wozu dynamische Speicherverwaltung?

Betrachtet man zum Beispiel die Datenstruktur

```
feld: ARRAY [0..999] OF INTEGER;
```

so werden hierfür `1000*2` Byte reserviert. Der Nachteil eines Feldes liegt darin, daß die Anzahl der Elemente im voraus festgelegt werden muß. Dies kann zu Speicherplatzverschwendung führen, wenn in einem Programm nicht das gesamte Feld belegt wird. Gravierender ist es jedoch, wenn während der Programmausführung, zum Beispiel durch Tastatureingabe, mehr Daten erzeugt werden, als vom Programmierer vorgesehen waren.

Abhilfe schafft hier das Zeigerkonzept. Man spricht von einer dynamischen Datenstruktur, da die Anzahl der zu belegenden Bytes erst während des Programmablaufs nach Bedarf reserviert wird. Bereits belegter, aber nicht weiter benötigter Speicherplatz kann wieder für neue Variablen freigegeben werden.

Der Umgang mit Zeigern

In Modula sind Zeiger immer an bestimmte Datentypen gebunden. Ein Pointertyp lautet »`POINTER TO` *<Basistyp>*«. Eine Typdeklaration lautet daher:

```
TYPE <PointerBezeichner> = POINTER TO <Basistyp>;
```

Ist nun die Variable

```
VAR p: <PointerBezeichner>;
```

gegeben, so »zeigt« `p` auf eine Variable vom Typ *<BasisTyp>*, das heißt, `p` enthält die Adresse einer Variablen vom Typ *<BasisTyp>*. Da `p` eine Adresse enthält, also beim Atari einen 32-Bit-Wert, beansprucht `p` nur 4 Byte. `p` selbst ist nun eine Variable (global oder lokal) und wird daher nicht auf dem Heap abgespeichert. Die Prozedur `Storage.ALLOCATE` reserviert nun Platz für eine Variable vom Typ *<BasisTyp>* auf dem Heap. Dies ist die Variable, auf die `p` »zeigt«, das heißt, deren Adresse `p` enthält.

Diese Variable auf dem Heap heißt `p^` (lies »p-*ceil*«, etwa dt. »p-Dach«; die abgewandelte deutsche Aussprache »p-Ziel« ist noch anschaulicher).

Beispiel:

```
MODULE Zeigerl;

FROM Storage IMPORT ALLOCATE;

VAR p: POINTER TO REAL;
```

```
BEGIN
  ALLOCATE(p, SIZE(p^));
  p^ := 1.0;
END Zeigerl.
```

Hier wird die Zahl 1.0 auf dem Heap abgelegt. `ALLOCATE` braucht dazu die Größe des Speicherbereiches, die für die betreffende Variable, die auf dem Heap reserviert werden soll. Man erhält diesen Wert mit `SIZE(p^)`.

Der Pfeil deutet an, daß `p` auf `p^` zeigt. An die Variable `p^` (also an unserem Wert 1.0) kommt man nur über `p` heran. Zum Beispiel schreibt

```
WriteReal(p^,3,1);
```

den Inhalt des Speichers, auf den `p` zeigt. Man erkennt, daß pro abgespeichertem Datum ein Zeiger existieren muß, um an die Daten heranzukommen. Das macht das ganze Verfahren vorläufig ineffizient. Von Dynamik kann auch keine Rede sein, da die Anzahl der Pointer wiederum vor der Laufzeit bekannt sein müßte. Der Trick besteht nun darin, daß man auch die Zeiger selbst auf dem Heap ablegt! Lediglich einzelne Zeiger, zum Beispiel auf das erste oder letzte abgelegte Element, werden separat vorreserviert. Dieses Verfahren erreicht man durch den `RECORD`-Typ:

```
TYPE Knoten = RECORD
                info: <IrgendeinTyp>;
                naechster: POINTER TO Knoten;
              END;
```

Hierbei enthält `info` die eigentlich abzuspeichernde Information und `naechster` die Adresse, an der die nächste (aus Information und Zeiger bestehende) zu finden ist.

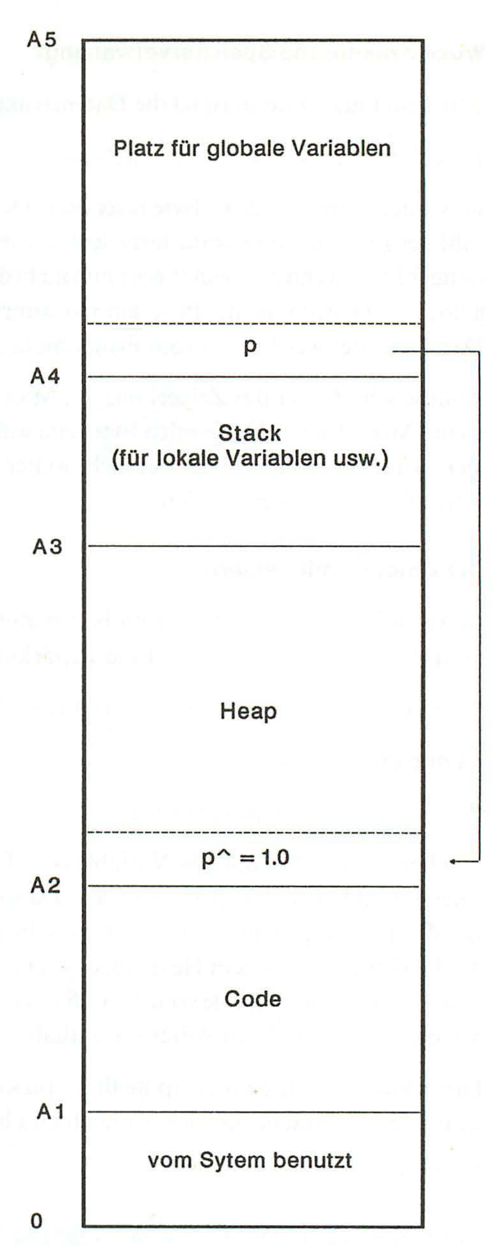


*Bild 1.21: Speicher mit p und p^*

Das nachfolgende Programm erzeugt eine »verkettete Liste« der abgebildeten Form.

Wir benötigen nur den globalen Zeiger `oben` um beliebig viele über die Tastatur hereinkommende `CARDINAL`-Zahlen abzuspeichern. Um die gesamte Liste wieder auszulesen, hangelt man sich von `Anfang` über die einzelnen Zeiger hinunter bis zum Ende. Das Ende ist dadurch markiert, daß das zuerst abgelegte Element, welches keinen Nachfolger hat, auf `NIL` zeigt. `NIL` ist ein vordefinierter Zeiger ins »Nichts«. Er ist zu allen Zeigertypen kompatibel und wird intern meist als Adresse `0` oder `-1` dargestellt. Daher ist der Zugriff

```
p := NIL;
p^ := <...>;
```

nicht erlaubt, man würde damit einen Speicherbereich überschreiben wollen, der eigentlich für andere Zwecke gedacht war. Das nächste Programmstück zeigt die typische Schleifenkonstruktion.

Bild 1.22: Verkettete Liste auf dem Heap

```
p := Anfang
WHILE p # NIL DO
  <p^.info bearbeiten>
  p := p^.naechster
END
```

Bei Feldern wäre das zu vergleichen mit einer Anweisung der Form:

```
FOR i := 0 TO max DO <feld[i] bearbeiten> END;
```

```
MODULE ZeigerDemo;

FROM Storage IMPORT ALLOCATE;
FROM InOut   IMPORT ReadCard, Read, WriteString, WriteLn; WriteCard;

TYPE
  Zeiger = POINTER TO Knoten;(* Vorwärtsreferenz bei Zeiger erlaubt! *)
  Knoten = RECORD
      zahl      : CARDINAL;                  (* zu speichernde Information *)
```

```
          naechster : Zeiger              (* Zeiger für die Verkettung  *)
          END;

VAR
   Anfang, p : Zeiger;
   taste     : CHAR;

BEGIN
  WriteString("Geben Sie fortlaufend nat. Zahlen ein (0 = Ende)"); WriteLn;
  Anfang: =NIL;
  REPEAT                                              (* Liste einlesen *)
    ALLOCATE(p,SIZE(p^));(* Speicherbereich für einen Knoten reservieren *)
    ReadCard(p^.zahl);
    p^.naechster := Anfang;
    Anfang := p
  UNTIL Anfang^.zahl = 0;
  WriteLn;                                            (* Liste auslesen *)

  p := Anfang;
  WHILE p  #  NIL DO
    WriteCard(p^.zahl,1); WriteLn;
    p := p^.naechster
  END;
  Read(taste)
END ZeigerDemo.
```

Schauen Sie sich genau an, wie die Liste aufgebaut wird und wie sie anschließend ausgelesen wird. Folgendes dürfte auffallen:

1. Die Zahlen werden in umgekehrter Reihenfolge ausgegeben wie sie eingelesen wurden.
2. In der Typdeklaration taucht der Typ `Knoten` in der Zeigerdeklaration auf, bevor er deklariert wird (das geschieht erst in der nächsten Zeile). Eine solche »Vorwärtsreferenz« ist bei Zeigern erlaubt.
3. Der Speicher wird nach der Ausgabe nicht wieder freigegeben.
4. Es können beliebig viele (jedenfalls so viele, wie auf den Heap passen) Zahlen gespeichert werden.
5. Man kommt nicht auf Anhieb an eine bestimmte Zahl heran. Möchte man zum Beispiel die dritte Zahl lesen, muß man sich erst durch die beiden ersten Elemente »durchhangeln«.

Zu 1:
Wir haben hier eine Datenstruktur, die im allgemeinen als Keller oder Stapel bezeichnet wird. Was man zuletzt »obendrauf« legt, muß man auch zuerst wieder abholen. Es gibt andere Strukturen wo der erste gespeicherte Wert auch wieder am Beginn geholt wird. Näheres dazu später im Abschnitt 2.2.2.

Zu 3:
Bei unserem kleinen Beispiel ist das nicht weiter tragisch, da der angeforderte Speicher nach Beendigung des Programms automatisch freigegeben wird. Wird aber innerhalb eines Programmes ständig neuer Speicher benötigt, so gibt man nicht mehr benötigten Speicher mit der Prozedur `DEALLOCATE` wieder frei.

Zu 4:
Genau das war ja das Ziel! Unser eingangs geschildertes Problem der unbekannten Indexgröße für ein Feld ist damit gelöst. Wenn man nicht vorher weiß, wieviele Feldelemente sich anhäufen, speichert man sie als Liste auf dem Heap.

Zu 5:
Das ist ein Nachteil. Bei einem Feld kommt man über den Index sofort an jedes Element. Listen kann man im Grunde nur der Reihe nach lesen.

Wenn Sie zum erstenmal von Zeigern gehört haben, bleiben sicherlich noch eine Menge Fragen, die nicht durch ein Beispielprogramm zu klären sind. Wir bringen hier aber keine weiteren Beispiele, da im gesamten Kapitel 2 mit dynamischen Datenstrukturen gearbeitet wird und dort systematisch alle Kunstgriffe vermittelt werden.

Dabei hat man es mit folgenden komplexen Strukturen zu tun:

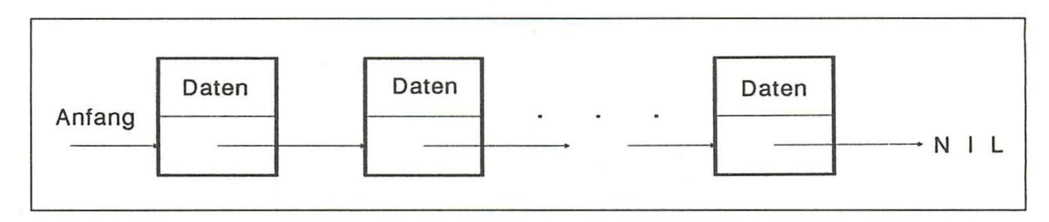


Bild 1.23: Verkettete lineare Liste

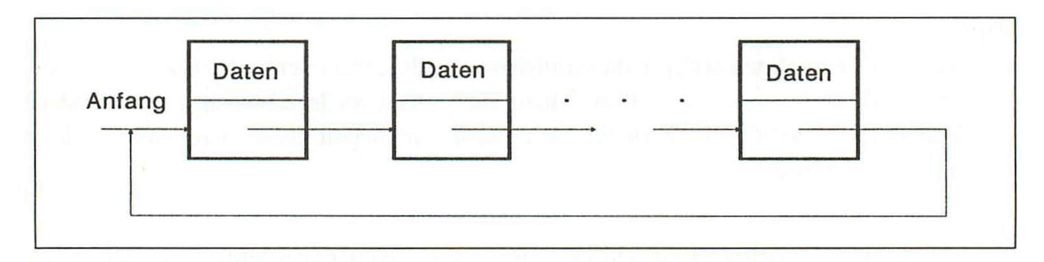


Bild 1.24: Ringliste

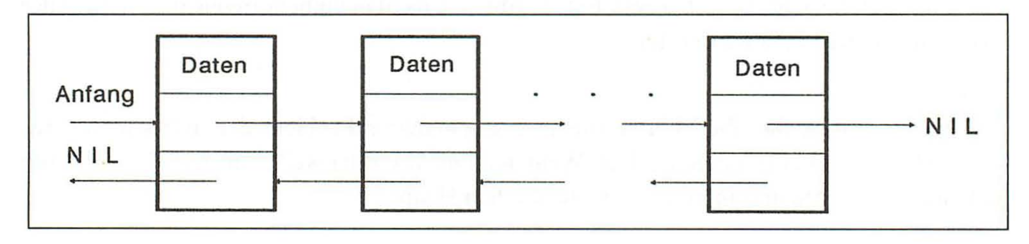


Bild 1.25: Doppelt verkettete Liste

Wurzel
Daten
links rechts
Daten
links rechts
Daten
links rechts

Bild 1.26: Binärer Baum

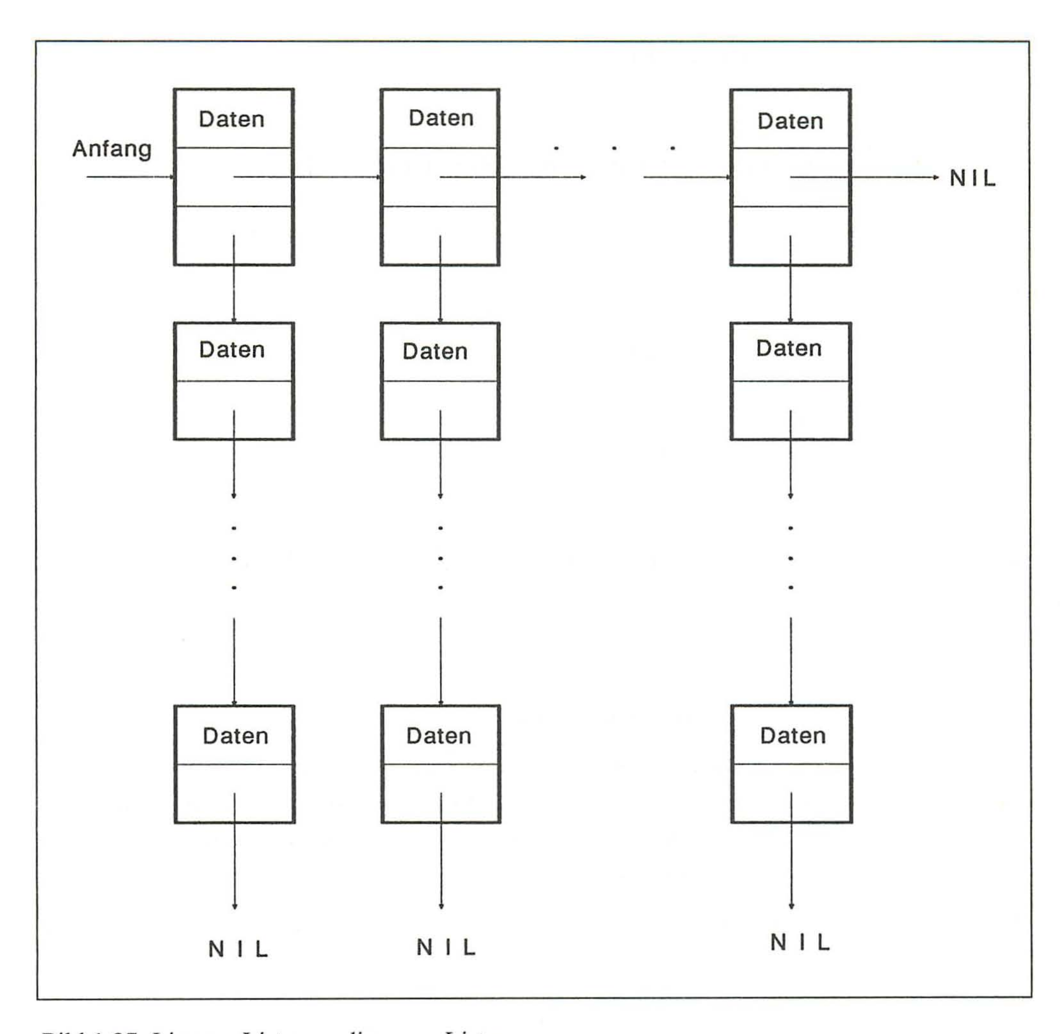


Bild 1.27: Lineare Liste von linearen Listen

Erlaubte Operationen mit Pointern

Zuweisung

Die Zuweisung ist nur zwischen Pointen mit gleichem Basistyp erlaubt:

```
p : = q
```

ist also falsch, falls `p` vom Typ `POINTER TO Typ1` und `q` vom Typ `POINTER TO Typ2` ist.

Vergleich

Zwischen Pointern gleichen Typs gibt es nur

`p = q` (Gleichheit) und

`p # q` (Ungleichheit).

Andere Vergleichsoperatoren sind für Zeiger nicht definiert.

Berechnungen

Verändern des Wertes eines Pointers geschieht im allgemeinen durch Zuweisung. Wichtig sind eigentlich nur drei Fälle:

`p := NIL;`	Zuweisen von `NIL`
`p := q;`	Zuweisen eines anderen Pointers
`ALLOCATE(p,p^);`	»Zuweisung« von Speicherplatz

Zuweisungen können natürlich auch über Funktion oder Parameterlisten erfolgen; damit verlagert sich die Zuweisung in die Prozedur. Andere Operationen sind nicht erlaubt, allerdings gestatten manche Compiler `INC` und `DEC` auf Pointern. Unkontrolliert angewendet führt das möglicherweise zu einem Programmabsturz.

Im Modul `SYSTEM` gibt es einen »Joker-Pointer-Typ« `ADDRESS`. Dieser »Pointer« kann jedem Zeigertyp zugewiesen werden und umgekehrt; außerdem ist er mit `LONGCARD` kompatibel. Nach

```
FROM SYSTEM IMPORT ADDRESS;
VAR a1,a2: ADDRESS;
    p1: POINTER TO <irgend ein Typ>;
    p2: POINTER TO <noch ein Typ>;
```

geht also

```
a1 := p1;
p2 := a1; (* Jetzt ist p1 = p2 *)
a1 := 123L * a2;
INC(a2);
DEC(a1,4);
```

Über den Sinn insbesondere der letzten drei Anweisungen läßt sich streiten. Wir zeigen allerdings in Kapitel 2.1 eine brauchbare Anwendung zur »POINTER-Arithmetik«.

Ein Selbsttest

Dies ist sicherlich neben der Rekursion der schwierigste Abschnitt gewesen. Deshalb sei ein kleiner Test gestattet.

Gegeben sei folgende verzeigerte Liste von »Knoten«. Es gilt die Deklaration:

```
TYPE Knoten = RECORD
                info: CHAR;
                naechster: POINTER TO Knoten
              END;
```

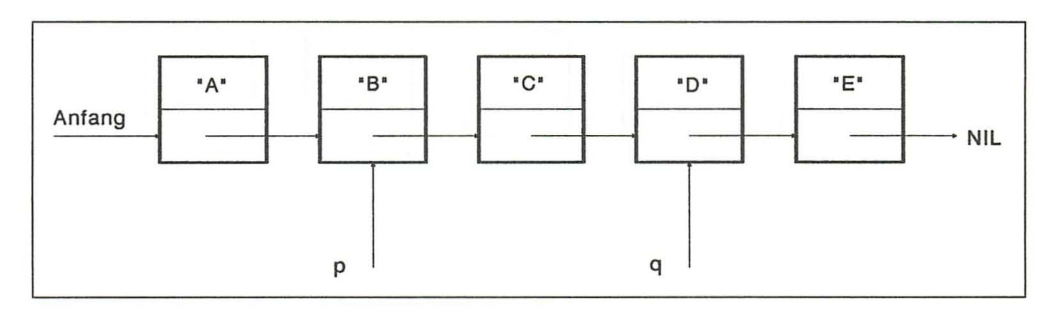


Bild 1.28: Eine lineare Liste mit den Inhalten »A«, »B«, »C«, »D«, »E«.

Frage:
Worin besteht der Unterschied in den Wertzuweisungen

a) `p := q;`
b) `p^.ch := q^.ch;`
c) `p^ := q^;`

Lösung:

`p := q`
bewirkt, daß der Zeiger `p` auf das selbe Element wie `q` zeigt. `p` zeigt jetzt auf den Knoten mit dem Inhalt "`D`".

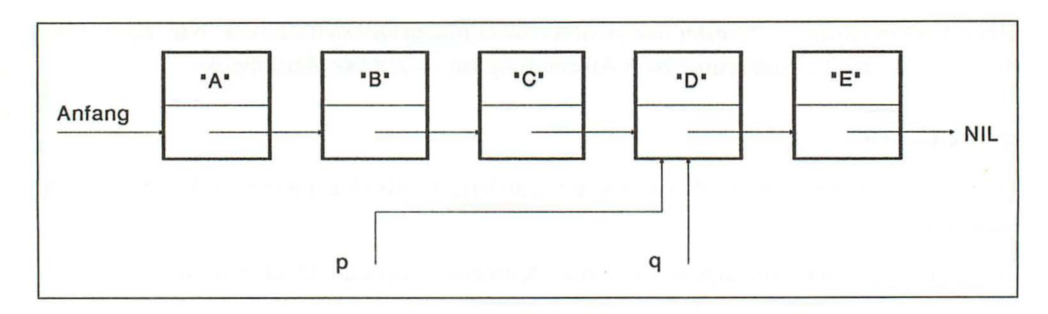


Bild 1.29: Liste nach p:=q

Nach `p^.ch := q^.ch`
wird "B" durch "D" überschrieben.

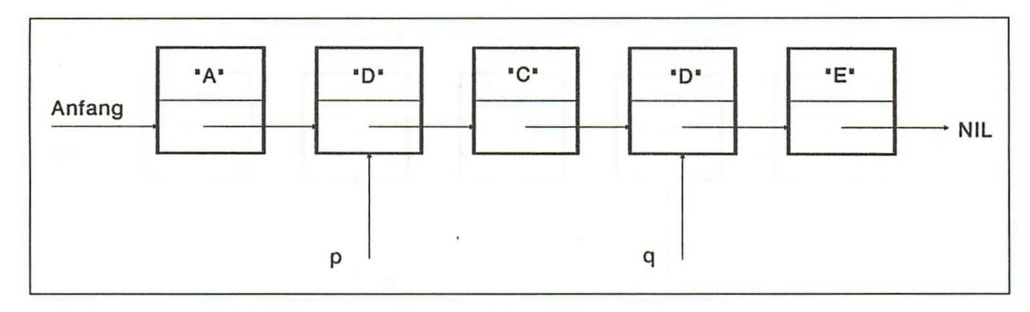


Bild 1.30: Liste nach p1.ch:=q1.ch

Nach `p^ := q^` wird der gesamte Knoten auf den `p` zeigt, überschrieben:

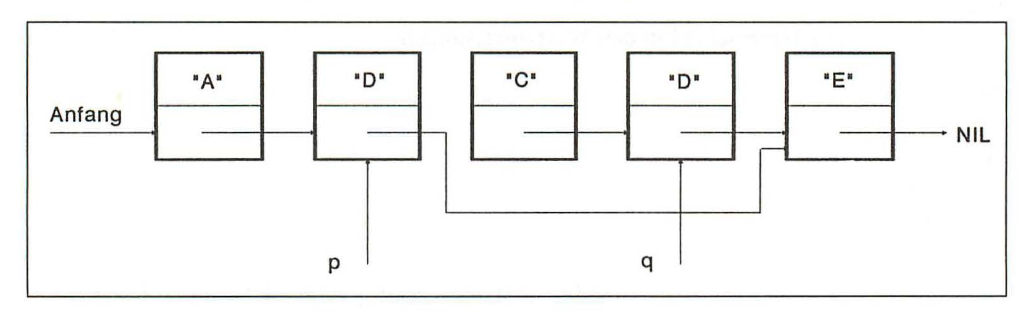


Bild 1.31: Liste nach p1:=q1

Die Knoten mit der »Information« "C" und "D" sind nun aus der Liste »ausgekettet«. "D" ist nur noch über q zugänglich, "C" überhaupt nicht mehr! Die folgende Leseschleife liefert nur »ADE«.

```
p := anfang;
WHILE p#NIL DO
  Write(p^.ch);
  p := p^.naechster
END;
```

An diesen Beispielen erkennt man, daß man mit Pointern sehr umsichtig arbeiten sollte!

1.6.7 Der Datentyp »PROZEDUR«

Schauen wir uns das Beispiel der Nullstellenbestimmung aus (1.5.4) an!

Hier wird eine Funktion f aufgerufen, die zuvor durch eine Funktionsprozedur deklariert wurde. Wenn man aber damit in einem Modul die Nullstellen mehrerer Funktionen bestimmen möchte, so ist dies nicht möglich, da die Prozedur Nullstelle mit der »globalen« Funktion f operiert. Man möchte gerne eine Funktion in der Parameterliste von Nullstelle mit übergeben können.

In Modula ist es möglich, beliebige Prozeduren – die selber wieder beliebige Parameterlisten haben können – als Parameter in einer anderen Prozedur zu verwenden! Hierzu dient der Datentyp PROCEDURE. Die Typdeklaration für die Funktion

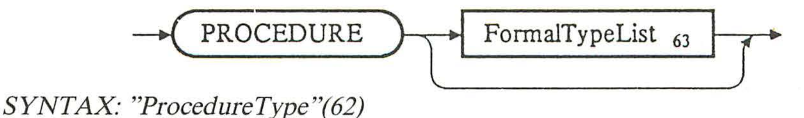
```
PROCEDURE f(x:REAL): REAL
```

lautet:

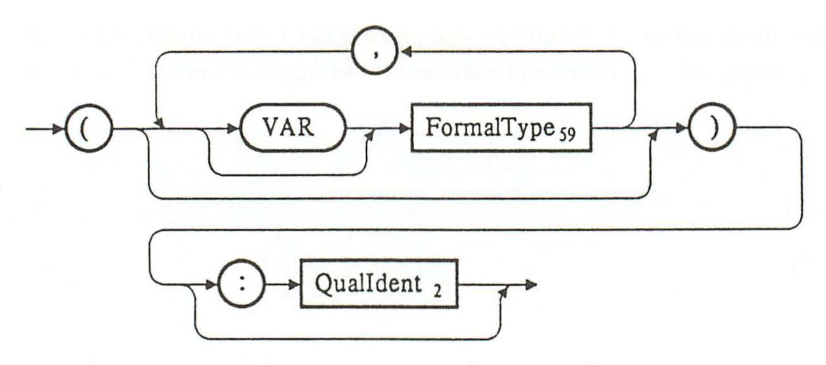
```
TYPE reelleFunktion = PROCEDURE(REAL):REAL;
```

Es wird also die Parameterliste ohne die Variablenbezeichner gebraucht.

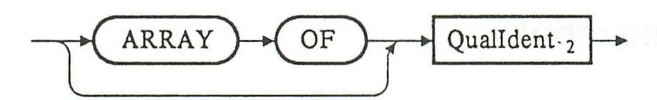
ProcedureType$_{62}$



SYNTAX: "ProcedureType"(62)



SYNTAX: "FormTypeList"(63)



SYNTAX: "FormType"(59)

Allgemein: Es lassen sich mit dem Typ `reelleFunktion` nun auch Variablen erklären wie

```
VAR f,g: reelleFunktion;
```

Nehmen wir an, wir haben eine Funktion

```
PROCEDURE DritteWurzel(wert:REAL): REAL;
BEGIN
  <...>
END DritteWurzel;
```

definiert. Dann kann man diese gesamte Funktion selbst (nicht das Ergebnis der Funktion!) den Variablen `f` oder `g` zuweisen:

```
f := DritteWurzel;
```

nun ist möglich

```
y := f(x);
```

was dasselbe leistet wie

```
y := DritteWurzel(x);
```

Man beachte den Unterschied zwischen den beiden Zuweisungen

```
g := f;
y := f(x);
```

im ersten Fall wird eine Funktion einer Funktionsvariablen zugewiesen, im zweiten Fall ein REAL-Wert, nachdem eine Funktion aufgerufen wurde. Das gleiche gilt für

```
g := DritteWurzel;           (* Zuweisung einer Funktion *)
y := DritteWurzel(x);        (* Zuweisung des Funktions-Wertes *)
```

nun wird klar, warum man beim Aufruf parameterloser Funktionen die Klammern für die leere Parameterliste mit angeben muß:

`f1 := f2;` weist eine gesamte Funktion zu.
`y := f2();` weist einen Funktionswert nach der Berechnung zu.

Dabei macht es keinen Unterschied, ob `f2` eine Funktion oder eine `PROCEDURE`-Variable ist.

Bleibt noch zu erwähnen, daß es für parameterlose Prozeduren einen eigenen Standard-Datentyp `PROC` gibt, den man sich so definiert vorstellen kann:

```
TYPE PROC = PROCEDURE;
```

Das folgende Programm greift die Nullstellenberechnung für verschiedene Funktionen auf. Weitere Beispiele findet man in Kapitel 2 und 5.

```
MODULE ProzeduraleParameterDemo;

FROM InOut     IMPORT ReadReal, WriteReal, WriteString, WriteLn, Read;
FROM MathLib0 IMPORT cos, ln;

TYPE Funktion = PROCEDURE(REAL) : REAL;

PROCEDURE f(x : REAL) : REAL;
BEGIN
  RETURN x - cos(x)
END f;

PROCEDURE g(x:REAL) : REAL;
BEGIN
  RETURN 3.0*x*x - 8.0*x + 4.0
END g;

PROCEDURE Nullstelle(fkt : Funktion; eps,a,b : REAL) : REAL;
VAR mitte, sign : REAL;
```

```
BEGIN
  IF fkt(a) > 0.0 THEN sign := -1.0 ELSE sign := 1.0 END;
  REPEAT
    mitte := (a+b) / 2.0;
    IF  sign*fkt(mitte) < 0.0 THEN a := mitte ELSE b := mitte
   UNTIL ABS(a-b) < eps;
   RETURN mitte
END Nullstelle;

VAR x1,x2,toleranz : REAL;
    taste          : CHAR;

BEGIN
  x1 := 0.0; x2 := 1.0; toleranz := 1.0E-10;
  WriteString("Die Nullstelle von 'f(x) = x - cos(x)      ' lautet: ");
  WriteReal(Nullstelle(f,toleranz,x1,x2),12,10); WriteLn;
  WriteString("Die Nullstelle von 'g(x) = 3*x*x - 8*x + 4' lautet: ");
  WriteReal(Nullstelle(g,toleranz,x1,x2),12,10); WriteLn;
  WriteString("Die Nullstelle von 'h(x) = ln(x)           ' lautet: ");
  WriteReal(Nullstelle(ln,toleranz,0.5,10.0),12,10); WriteLn;
  Read(taste)
END ProzeduraleParameterDemo.
```

1.6.8 Typgleichheit, Ausdrucks- und Zuweisungs-Kompatibilität

Sicherlich haben Sie sich beim Programmieren über so manche Compiler-Fehlermeldung geärgert, die mit der Kompatibilität zu tun hatte. Man muß in Modula folgendes zur Kenntnis nehmen:

Typgleichheit

Zwei Variablen `x1` und `x2` mit den Typen `T1` und `T2`:

```
VAR x1: T1; x2: T2;
```

werden als vom selben Datentyp bezeichnet, wenn

- `T1` und `T2` derselbe Name ist, zum Beispiel `CARDINAL`, `MeinTyp`; dies ist insbesondere der Fall, wenn `x1` und `x2` in einer einzigen Variablendeklaration stehen:
  `VAR x1,x2:T1;`
- `T1` und `T2` werden in einer Typdeklaration als synonym erklärt, also mit `TYPE T1=T2;` zum Beispiel:

```
TYPE
    str20 = ARRAY [1..19] OF CHAR;
    Zeichenkette = str20;
VAR
    x1: str20;
    x2: Zeichenkette;
```

- x1 und x2 sind Variablen desselben Aufzählungstyps.

Die folgenden Variablen x1 und x2 sind jedoch von verschiedenem Typ:

```
TYPE
    T1 = ARRAY [0..99] OF CARDINAL;
    T2 = ARRAY [0..99] OF CARDINAL;
VAR
    x1: T1;
    x2: T2;
```

Die Zuweisung x1:=x2 ist nun nicht möglich! Eine solche Konstruktion benutzt man nur, um eventuell später einen Typ umzudefinieren; das Programm läuft dann trotzdem.

```
TYPE
    T1 = ARRAY [0..99] OF REAL;
    T2 = ARRAY [0..99] OF CARDINAL;
```

Ausdruckskompatibilität

Zwei Operanden x1 und x2 vom Typ T1 und T2 können nur in einem Ausdruck verknüpft werden (zum Beispiel x1+x2) wenn gilt:

- T1 und T2 sind typgleich.
- T1 ist ein Unterbereichstyp mit Basistyp T2 (oder umgekehrt).
- T1 ist INTEGER oder CARDINAL und x2 eine Konstante oder vom Typ [min..max]mit 0 ≦ min,max ≦ MAX(INTEGER).
- T1 ist ein beliebiger POINTER-Typ und x2 die Konstante NIL.

Zuweisungskompatibilität

Es seien wieder `x1` und `x2` vom Typ `T1` und `T2`. Dann ist die Zuweisung `x1:=x2` erlaubt, falls gilt:

- `x1` und `x2` sind ausdruckskompatibel.
- `T1` ist `INTEGER` oder ein Unterbereich davon und `T2` ist `CARDINAL` oder ein Unterbereich davon (oder umgekehrt). Man hat bei der Zuweisung aber darauf zu achten, daß der Wert von `x2` im Wertebereich von `T1` liegt.
- `T1 = ARRAY <Bereich> OF CHAR` mit n-Elementen und `x2` ist eine String-Konstante mit höchstens n-Zeichen. Bei dieser Zuweisung werden die verbleibenden Zeichen mit `0C` aufgefüllt. Zum Beispiel:

```
TYPE String20 = ARRAY [0..19] OF CHAR;
VAR s: String20;
<...>
s := "Hallo";
```

1.7 Das Modulkonzept

Im folgenden wird eine Idee beschrieben, die der Sprache den Namen gab: **MODU**lar **LA**nguage (= »Modulare Sprache«). Es geht dabei um »lokale« Module die innerhalb von anderen Modulen definiert sind, sowie »externe Module«, die getrennt übersetzt werden können.

Das Konzept ist ein mächtiges Werkzeug zum Schreiben großer Programme. Bevor wir auf die Einzelheiten eingehen, zunächst die grundlegenden Ideen:

1.7.1 Das Geheimnisprinzip

»Ach wie gut, daß niemand weiß, daß ich Rumpelstilzchen heiß!« Das ist vielleicht ein unschöner Charakterzug, beim Programmieren aber guter Stil. Für Variablen in Prozeduren gilt: »Soviel lokal wie möglich, so wenig global wie nötig!« Das folgende – hoffentlich abschrekkende – Beispielprogramm unterstreicht diese These:

```
VAR i: CARDINAL;

PROCEDURE doppelt(n:CARDINAL): CARDINAL;
BEGIN
  INC(i);
  RETURN 2*n;
END doppelt;
```

Diese Prozedur wird einwandfrei kompiliert. Neben dem, was ihr Name besagt, nämlich den eingegebenen Wert zu verdoppeln, bastelt sie noch an der globalen Variablen `i` herum. Na und?

Betrachten wir nun die Ausdrücke

`doppelt(5)+i` und `i+doppelt(5)`

Wenn vorher gilt `i = 3`, so wird eventuell (je nach Compiler) der

erste Ausdruck: `2*5+(3+1) = 14` und der
zweite Ausdruck: `3+2*5 = 13`

So etwas ist schwer zu durchschauen, man erwartet bei einer Summe normalerweise Vertauschbarkeit der Summanden! Schlimm ist auch `doppelt(i)`. Das Ergebnis ist nicht *2*i*, sondern *2*i+1*!

Man spricht hierbei von »Seiteneffekten«. Die Prozedur `doppelt` greift auf die globale Variable `i` zu. Besser ist es, wenn eine Prozedur nur über eine klar definierte »Schnittstelle«, und das sollte nur die Parameterliste sein, kommuniziert. Weitere Variablen sollten lokal sein. Sie sind nach außen nicht sichtbar, werden also vor dem übrigen Programm geheim gehalten.

Modula treibt dieses Konzept noch weiter, in dem ganze Module, also Zusammenfassungen von Konstanten, Typen, Variablen, Prozeduren und Anweisungsteil »lokal« sein können. Dem »aufrufenden« übergeordneten Modul wird nur das mitgeteilt, was dieser wissen muß. Man spricht hierbei vom »Export« von Bezeichnern. Der Rest bleibt verborgen.

Das hat für den Programmierer den Vorteil, daß er sich auf das Wesentliche konzentrieren kann. Die Details interessieren nicht. »was ich nicht weiß, macht mich nicht heiß«. Die Krönung dieser Idee ist der sogenannte »opake Export«, bei der sogar der Datentyp verheimlicht wird.

Wir besprechen nun die Einzelheiten. Zunächst stellen wir das Konzept der lokalen Module vor (1.7.2). Danach zeigen wir, wie man selbst externe Module schreibt, die getrennt übersetzt werden können (1.7.3). Anschließend gehen wir auf Standardmodule ein, das sind solche Module, die bei jedem Modula-System mitgeliefert werden. Einige kennen Sie schon: `InOut` und `Storage`.

1.7.2 Lokale Module

Gegeben ist folgendes Schema:

```
MODULE Haupt;
<...>
  MODULE Unter;
  <...>
  END Unter;
BEGIN (* von 'Haupt' *)
  <...>
END Haupt.
```

Unter ist hier ein lokaler Modul, also ein eigenständiger Modul innerhalb eines anderen Moduls. Er wird in dessen Deklarationsteil aufgeführt und hat die Form:

```
MODULE <Untermodul-Name>
  <Importlisten>
  <Exportliste>
  <Deklarationsteil>
BEGIN
  <Anweisungsteil>
END <Untermodul-Name>;
```

Für lokale Module gelten die gleiche Sichtbarkeitsregeln wie für Prozeduren, bis auf folgende Abweichungen:

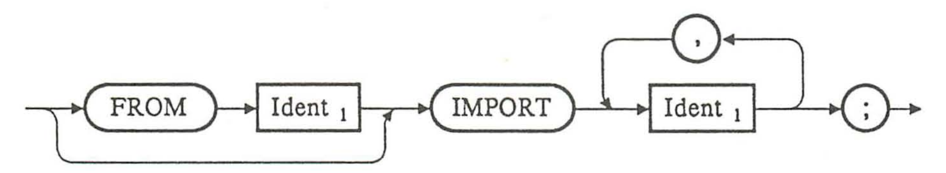
1. Ein Bezeichner eines inneren Moduls kann auch dem umschließenden Modul sichtbar gemacht werden, indem er in der EXPORT-Liste des inneren Moduls aufgeführt wird.
2. Bei Prozeduren sind immer auch die Bezeichner der Umgebung sichtbar, und zwar alle, die zu ihr »global« sind. Für Module gilt dies nicht! Module sind abgeschlossene Einheiten. Die außerhalb liegenden Bezeichner sind in ihnen nicht sichtbar. Soll ein Bezeichner auch innerhalb dieses Moduls sichtbar sein, so muß er importiert werden; das heißt, er muß in der IMPORT-Liste auftauchen.

Ein lokaler Modul ist also ein in sich abgeschlossenes Programmstück, daß mit seiner Umgebung nur über Import- und Exportlisten kommuniziert. Für den Import gelten folgende Besonderheiten:

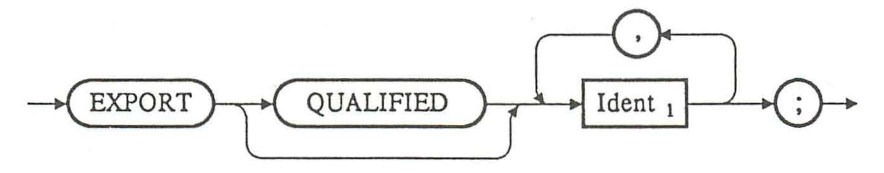
- Es dürfen auch nur die zu diesem Modul globalen Bezeichner importiert werden. Unter anderem kann ein lokaler Modul keine Prozeduren aus `InOut` importieren, wenn diese nicht dem umschließenden Modul bekannt sind.

- Wenn ein Verbund importiert wird, sind dem Modul auch dessen sämtliche Komponenten bekannt.
- Wenn ein Aufzählungstyp importiert wird, kennt der Modul auch alle Konstanten dieses Types. Man kann aber Konstanten eines Aufzählungstyp einzeln importieren. Dann sind allerdings weder dieser Typ noch die anderen Konstanten dieses Typs bekannt.

Für die Import- und Exportlisten gelten folgende Diagramme:



SYNTAX: "Import"(65)



SYNTAX: "Export"(66)

Wenn eine EXPORT-Liste das Schlüsselwort `QUALIFIED` enthält, muß der Bezeichner im umgebenden Modul qualifiziert benutzt werden. Das heißt, der Modulname wird mit einem Punkt vorangestellt (*<Modulname>.<Bezeichner>*). Zum Beispiel:

```
MODULE Haupt;
IMPORT InOut;
MODULE Unter1;
  EXPORT QUALIFIED a;
  CONST a = 5;
BEGIN
END Unter1;

MODULE Unter2;
  EXPORT Qualified a;
  CONST a = 2;
BEGIN
END Unter2;

BEGIN (* Haupt *)
  InOut.WriteCard(Unter1.a + Unter2.a, 6);
END Haupt.
```

Die Ausführung der Anweisungsteile von lokalen Modulen geschieht in der Reihenfolge, in der sie vom Compiler gelesen werden! Das folgende extreme Beispiel zeigt den Import und Export von Daten und Prozeduren in extremer Weise. Vielleicht nehmen Sie sich die Zeit, nachzuvollziehen, daß es lediglich »MODULA2« auf Bildschirm schreibt. Sämtliche anderen lokale Module sind wesentlich einfacher.

```
MODULE LokaleModuleTest;

FROM InOut IMPORT Write, Read, WriteCard;

VAR n : CARDINAL;
(*------------------------------------------------------------------------*)
MODULE Nix ;                                         (* Lokaler Modul Nix *)
  EXPORT BringeU, U;
  VAR U : CHAR;
  PROCEDURE BringeU : CHAR;
    BEGIN
     RETURN U               (* U wird im Anweisungsteil von O initialisiert *)
    END BringeU;
        (*  in Nix sind nur die lokalen Objekte 'U' und 'BringeU' sichtbar*)
END Nix;
(*------------------------------------------------------------------------*)
MODULE M;                                              (* Lokaler Modul M *)
  IMPORT Write, n;
  EXPORT mal;
  CONST x = 2;
  PROCEDURE mal(VAR c : CARDINAL);
    BEGIN
       c := x*n
    END mal;
  BEGIN Write('M')                             (* M erstes Ausgabezeichen *)
END M;
     (* in M sind 'n' und 'Write' sowie die Objekte 'x' und 'mal' sichtbar *)
(*------------------------------------------------------------------------*)
MODULE O;                                              (* Lokaler Modul O *)
  IMPORT Write, n, U;
  BEGIN
     n := 1; U := "U";                        (* n und U werden initialisiert *)
     Write("O")                                    (* O wird ausgegeben *)
END O;
(*                            in 'O' sind 'n', 'U', und Write sichtbar *)
(*------------------------------------------------------------------------*)
VAR d,taste : CHAR;
```

```
BEGIN
   d := 'D'; Write(d);                           (* 'D' wird ausgegeben *)
   Write(BringeU());
   Write("L"); Write("A");
   mal(n); WriteCard(n,1);                       (* '2' wird ausgegeben *)
   Read(taste)
           (* Hier sind 'Write', 'Read', 'WriteCard', 'n' ,'d' ,'taste',
                                  'BringeU','U' und 'mal' sichtbar  *)
END LokaleModuleTest.
```

1.7.3 Benutzerdefinierte externe Module

Jetzt geht es erst richtig los! Lokale Module ermöglichen zwar abgeschlossene Programmiereinheiten, sie müssen aber zusammen mit den übergeordneten Modulen übersetzt werden.

Wesentlich stärker ist das Konzept der externen oder äußeren Module, die für sich getrennt entwickelt, getestet und kompiliert werden können. Dadurch läßt sich einerseits die Arbeit an einem großen Programmpaket segmentieren und auf mehrere Programmierer verteilen. Zudem können diese Module auch von mehreren Programmen benutzt werden. Sie kennen solche Module bereits vom Lieferumfang Ihres Systems her. Jetzt sollen Sie lernen, selber solche Module zu schreiben.

Ein solcher externer Modul besteht aus 2 Teilen:

- dem *Definitionsmodul* und
- dem *Implementationsmodul*

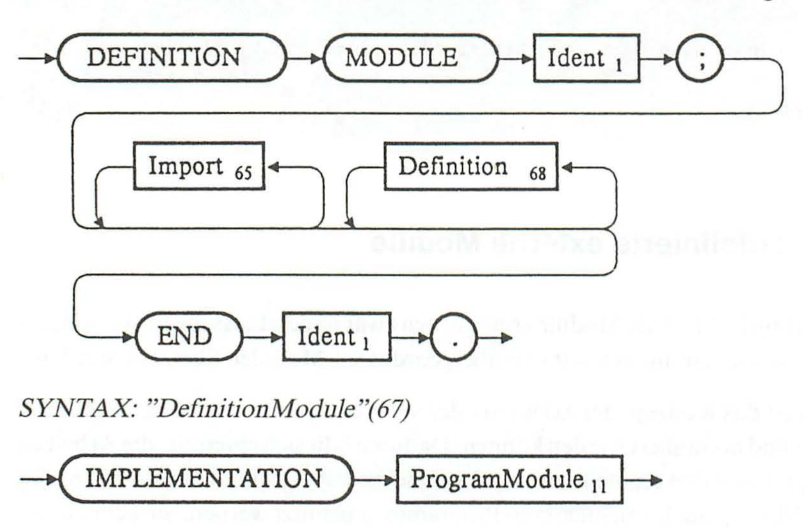
Im Definitionsmodul werden sämtliche Konstanten, Typen, Variablen und Prozeduren aufgeführt, die auch anderen Modulen zugänglich gemacht werden sollen. Eine EXPORT-Liste erübrigt sich deshalb. Bei Prozeduren ist nur der Prozedurkopf aufzuführen, nicht aber der Rumpf.

Im Implementationsmodul werden die Prozeduren vollständig aufgeführt. Hier erfolgt also die eigentliche Ausformulierung. Zusätzlich kann ein Implementationsmodul weitere Definitionen und Hilfsprozeduren enthalten, die für die Implementation nötig sind, die aber von der Außenwelt nicht zu benutzen sind.

Konstanten, Typen und Variablen, die im Definitionsmodul deklariert sind, werden im Implementationsmodul nicht noch einmal aufgeführt.

Aus diesem Grunde muß immer zuerst der Definitionsmodul übersetzt werden, dann der zugehörige Implementationsmodul. Hierdurch sind letzterem die Deklarationen bekannt. Nach erfolgreicher Übersetzung können dann die im Definitionsmodul aufgeführten Bezeichner in

anderen Modulen genutzt werden. Hierzu sind sie in deren Importliste aufzuführen. Braucht ein externer Modul Bezeichner aus anderen Modulen, so sind diese natürlich von ihm zu importieren. Dieser Import kann sowohl im Implementationsmodul erfolgen (wenn die Bezeichner nur für die Implementation gebraucht werden) als auch im Definitionsmodul (wenn Bezeichner aus anderen Modulen bereits im Definitionsmodul benötigt werden).



SYNTAX: "DefinitionModule"(67)

SYNTAX: "ImplementationModule"(68)

Im folgenden bringen wir einige sinnvolle Beispiele, die Leistungen bereitstellen, die Sie in Ihren Programmen nutzen können. Im ersten Beispiel geht es um Berechnungen rund um das Kalenderdatum.

Rund um das Datum

`DatumBib` liefert 2 Datentypen und 14 Prozeduren. Wahrscheinlich sind Sie nun auf eine breite Erörterung über die Leistung des Moduls gefaßt. Wir dürfen Sie aber beruhigen, es kommt nichts dergleichen. Das ist auch gut so, denn die Leistungsbeschreibung soll voll aus dem Definitionsmodul hervorgehen.

```
DEFINITION MODULE DatumBib;

TYPE Str10    = ARRAY [0..9] OF CHAR;
     DatumTyp = RECORD
                  tag   : [1..31];
                  monat : [1..12];
```

```
                  jahr  : CARDINAL
                END;

PROCEDURE heute(VAR datum : DatumTyp);
  (* Holt das Atari-Datum *)
PROCEDURE SchaltJar(jahr : CARDINAL) : BOOLEAN;
  (* Ist das Jahr ein Schaltjahr ? *)
PROCEDURE MonatLaenge(datum : DatumTyp) : CARDINAL;
  (* Liefert die Anzahl der Tage des Monats 'datum.monat' *)
PROCEDURE TagImJahr(datum : DatumTyp) : CARDINAL;
  (* Der wievielte Tag ist 'datum' im Jahr ? *)
PROCEDURE SchaltJahreSeit0(Jahr : CARDINAL) : CARDINAL;
  (* Anzahl der Schaltjahre seit dem 01.01.00 *)
PROCEDURE TageSeit0(datum : DatumTyp) : LONGCARD;
   (* Berechnet die Anzahl der Tage seit dem 01.01.00 *)
PROCEDURE TagAbstand(dat1,dat2 : DatumTyp) : LONGINT;
  (* Berechnet die Differenz zweier Daten in Tagen *)
PROCEDURE TagInWoche(datum : DatumTyp) : CARDINAL;
  (* 1 = Montag, 2 = Dienstag, .. ,7 = Sonntag *)
PROCEDURE NaechstTag(VAR datum : DatumTyp);
  (* Liefert das Datum des nächsten Tages *)
PROCEDURE VorTag(VAR datum : DatumTyp);
  (* Liefert das Datum des vergangenen Tages *)
PROCEDURE JahrHat53Wochen(Jahr : CARDINAL) : BOOLEAN;
  (* Hat das Jahr die 53. Kalenderwoche ? *)
PROCEDURE KalenderWoche(datum : DatumTyp) : CARDINAL;
  (* In der wievielten Kalenderwoche liegt 'datum'? *)
PROCEDURE DatumZuString(datum : DatumTyp; VAR s : Str10);
   (* Wandelt ein Datum in einen String der Form tt.mm.jjjj um. *)
PROCEDURE WochenTag(datum : DatumTyp; VAR WTag : Str10);
   (* Ermittelt den Tagesnamen eines Datums, z.B WTag = "Montag" *)
END DatumBib.
```

Soweit die Schnittstellenbeschreibung des Moduls. Es fehlt noch die Implementation. Möglicherweise sind sie von ihrer Länge unangenehm überrascht. Wer sich aber den Programmtext kurz anschaut, wird erkennen, das jede Prozedur für sich genommen sehr einfach ist. Die Gesamtlänge kommt nur durch die Vielzahl der Prozeduren zustande, steht also im Verhältnis zur Leistung des Moduls. Bis auf die Prozedur `heute` ist alles selbsterläuternd. Für deren Ausformulierung muß man das Datum von der Atari-Uhr lesen. Hierzu liefert der Modul `GEMDOS` die Prozedur `GetDate`. Man braucht das Datum nur noch zu dekodieren. Beachten Sie, das `GEMDOS` lediglich für den Implementationsmodul importiert wird, nicht aber im Definitionsmodul. Schließlich geht es den Benutzer nichts an, wie wir unsere Aufgaben erledigen.

```
IMPLEMENTATION MODULE DatumBib;

FROM GEMDOS IMPORT GetDate;
PROCEDURE heute(VAR datum : DatumTyp);
VAR
    DatumCode : CARDINAL;
BEGIN
  GetDate(DatumCode);
  (* Datum kodiert als ((jahr-1900)*16 + monat)*32 + tag *)
  WITH datum DO
    tag := DatumCode MOD 32;
    DatumCode := DatumCode DIV 32;
    monat := DatumCode MOD 16;
    DatumCode := DatumCode DIV 16;
    jahr := DatumCode + 1980
  END
END heute;

PROCEDURE SchaltJahr(jahr : CARDINAL) : BOOLEAN;
BEGIN
  RETURN (jahr MOD 4 = 0) & (NOT(( jahr MOD 100 = 0) & (jahr MOD 400 # 0)))
END SchaltJahr;

PROCEDURE MonatLaenge(datum : DatumTyp) : CARDINAL;
BEGIN
  CASE datum.monat OF
    4,6,9,11 : RETURN 30 |
    2        : IF SchaltJahr(datum.jahr) THEN RETURN 29 ELSE RETURN 28 END
  ELSE RETURN 31 END
END MonatLaenge;

PROCEDURE TagImJahr(datum : DatumTyp) : CARDINAL;
VAR
    tage, monat,EndMonat : CARDINAL;
BEGIN
  EndMonat := datum.monat - 1;
  datum.monat := 1;
  tage := 0;
  FOR monat := 1 TO EndMonat DO
      datum.monat := monat;
      INC(tage,MonatLaenge(datum))
  END;
  RETURN tage + datum.tag
END TagImJahr;
```

```
PROCEDURE SchaltJahreSeit0(Jahr : CARDINAL) : CARDINAL;
BEGIN
  DEC(Jahr);
  RETURN (Jahr DIV 4) - (Jahr DIV 100) + (Jahr DIV 400)
END SchaltJahreSeit0;

PROCEDURE TageSeit0(datum : DatumTyp) : LONGCARD;
BEGIN
  RETURN 365L*LONG(datum.jahr)
        + LONG(SchaltJahreSeit0(datum.jahr) + TagImJahr(datum))
END TageSeit0;

PROCEDURE TagAbstand(dat1,dat2 : DatumTyp) : LONGINT;
BEGIN
  RETURN LONGINT(TageSeit0(dat1)) - LONGINT(TageSeit0(dat2))
END TagAbstand;

PROCEDURE TagInWoche(datum : DatumTyp) : CARDINAL;
VAR
    hilf : CARDINAL;
BEGIN
  hilf := SHORT((TageSeit0(datum) - 1L ) MOD 7L );
  IF hilf = 0 THEN RETURN 7 ELSE RETURN hilf END
END TagInWoche;

PROCEDURE NaechstTag(VAR datum : DatumTyp);
BEGIN
  WITH datum DO
    IF tag = MonatLaenge(datum) THEN
      tag := 1;
      monat := monat MOD 12 + 1;
      IF monat = 1 THEN INC(jahr) END
    ELSE INC(tag) END
  END
END NaechstTag;

PROCEDURE VorTag(VAR datum : DatumTyp);
BEGIN
  WITH datum DO
    IF tag = 1 THEN
      IF monat = 1 THEN DEC(jahr); monat:=12 ELSE DEC(monat) END;
      tag := MonatLaenge(datum)
    ELSE DEC(tag) END
  END
```

```
END VorTag;

PROCEDURE JahrHat53Wochen(Jahr : CARDINAL) : BOOLEAN;
VAR Neujahr : DatumTyp;
BEGIN
  WITH Neujahr DO tag := 1; monat := 1; jahr:=Jahr + 1 END;
  RETURN TagInWoche(Neujahr) IN {5,6}
END JahrHat53Wochen;

PROCEDURE KalenderWoche(datum : DatumTyp) : CARDINAL;
VAR Neujahr       : DatumTyp;
    hilf,KaWoche : CARDINAL;
BEGIN
  WITH Neujahr DO tag:=1; monat:=1; jahr:=datum.jahr END;
  hilf := TagInWoche(Neujahr);
  KaWoche := (TagImJahr(datum) + hilf - 2 ) DIV 7;
  IF hilf < 5 THEN INC(KaWoche) END;
  IF (KaWoche = 53) & NOT(JahrHat53Wochen(datum.jahr)) THEN KaWoche:=1
    ELSIF KaWoche=0 THEN VorTag(Neujahr); KaWoche:= KalenderWoche(Neujahr) END;
  RETURN KaWoche;
END KalenderWoche;

PROCEDURE DatumZuString(datum : DatumTyp; VAR s : Str10);
VAR
    pos : CARDINAL;
BEGIN
  s[2]:="."; s[5]:=".";                              (* s = "__.__.____" *)
  WITH datum DO
    s[0] := CHR(tag   DIV 10 + ORD("0")) ; s[1] := CHR(tag   MOD 10 + ORD("0"));
    s[3] := CHR(monat DIV 10 + ORD("0")) ; s[4] := CHR(monat MOD 10 + ORD("0"));
    FOR pos := 9 TO 6 BY -1 DO
      s[pos] := CHR(jahr MOD 10 + ORD("0")) ;
      jahr := jahr DIV 10
    END;
  END;
END DatumZuString;

PROCEDURE WochenTag(datum : DatumTyp; VAR WTag : Str10);
BEGIN
  CASE TagInWoche(datum) OF
    1 : WTag := "Montag"     |
    2 : WTag := "Dienstag"   |
    3 : WTag := "Mittwoch"   |
    4 : WTag := "Donnerstag"|
```

```
      5 : WTag := "Freitag"   |
      6 : WTag := "Samstag"   |
      7 : WTag := "Sonntag"   |
    END
  END WochenTag;

END DatumBib.
```

Zum Austesten des Moduls bringen wir ein kleines Demonstrationsprogramm, was einen großen Teil der Prozeduren aufruft. An der Importliste erkennen Sie, das sich die Benutzung eines selbstverfaßten Moduls in keiner Weise von der des Modula-Systems unterscheidet.

Noch ein Hinweis zur Handhabung: Wenn Sie nicht das Megamax-System benutzen, benennen Sie zunächst den Filenamen DATUMBIB.D in DATUMBIB.DEF und DATUMBIB.I in DATUMBIB.MOD um (das müssen sie bei allen Definitions- bzw. Implementationsmodulen tun). Übersetzen Sie anschließend den Definitionsmodul, dann den Implementationsmodul. Sorgen Sie dafür, daß diese Objekt-Dateien in einem Ordner landen, den der Compiler bei der anschließenden Übersetzung des Hauptmoduls DATUM.MOD findet. Hierzu sind die Suchpfade des Systems geeignet voreinzustellen (siehe Handbuch). Nennen Sie diesen Ordner einfach OBJEKTE. Dort sollten sich im Laufe der weiteren Lektüre dieses Buches sämtliche Übersetzungen der externen Module sammeln. Dies ist wichtig, da im folgenden andere Module auf diesen Modulen aufbauen.

Megamax-Benutzer finden alle Übersetzungen der externen Module im Ordner OBJEKTE. Machen Sie Ihrem Compiler diesen Ordner bekannt (Suchpfad in SHELL.INF angeben) und beginnen Sie gleich mit der Übersetzung von DATUM.M.

```
MODULE DatumDemo;

FROM InOut    IMPORT WriteLn, WriteString, WriteCard, Write, WriteInt,
                     Read, ReadCard;
FROM DatumBib IMPORT DatumTyp, Str10, heute, SchaltJahr, MonatLaenge,
                     TagImJahr, TagAbstand, TagInWoche, KalenderWoche,
                     NaechstTag, VorTag, DatumZuString, WochenTag;

PROCEDURE SchreibDatum(datum : DatumTyp);
VAR DatumStr, TagName : Str10;
BEGIN
  WochenTag(datum, TagName);
  WriteString(TagName); WriteString(", der ");
```

```
  DatumZuString(datum,DatumStr);
  WriteString(DatumStr); WriteLn;
END SchreibDatum;

VAR datum, datum2 : DatumTyp;
    ch            : CHAR;
    t, m, j       : CARDINAL;

BEGIN
  heute(datum);
  WriteString("Heute ist "); SchreibDatum(datum);
  WriteString("Es ist der ");WriteCard(TagImJahr(datum),1);
  WriteString(". Tag in diesem Jahr."); WriteLn;
  WriteString("Wir befinden uns in der "); WriteCard(KalenderWoche(datum),1);
  WriteString(". Kalenderwoche."); WriteLn;
  WriteString("Dieser Monat hat ");
  WriteCard(MonatLaenge(datum),1);  WriteString(" Tage."); WriteLn;
  WriteString("Dieses Jahr ist ");
  IF NOT SchaltJahr(datum.jahr) THEN Write("k") END;
  WriteString("ein Schaltjahr."); WriteLn;
  NaechstTag(datum); WriteString("Morgen ist "); SchreibDatum(datum);
  VorTag(datum);
  WriteLn; WriteString("Geben Sie ein zweites Datum ein :"); WriteLn;
  REPEAT
    REPEAT WriteString("Tag:   "); ReadCard(t) UNTIL (0 < t) & (t < 32);
    REPEAT WriteString("Monat: "); ReadCard(m) UNTIL (0 < m) & (m < 13);
    WriteString("Jahr:  "); ReadCard(j);
    WITH datum2 DO tag := t; monat := m; jahr := j END;
  UNTIL t <= MonatLaenge(datum2);                (* das Datum ist gültig *)
  WriteString("Abstand zwischen heute und dem eingegebenen Datum in Tagen: ");
  WriteInt(TagAbstand(datum2,datum),1);
  Read(ch)
END DatumDemo.
```

Externer Modul für komplexe Arithmetik

Das folgende Beispiel ist etwas mathematischer.

Es handelt sich um die Implementierung von Grundrechenarten bei komplexen Zahlen. Wir benötigen diesen Modul im 4. Kapitel bei der Erzeugung von Grafiken der »Mandelbrot-Menge« (auch als »Apfelmännchen« bekannt) und der »Julia- Mengen«.

Wenn Ihnen die zugrundeliegenden mathematischen Sachverhalte nicht bekannt sind, hier eine kurze Einführung.

Gegeben sei im Koordinatenkreuz ein Pfeil vom Nullpunkt zum Punkt *(3, 2)*. Wir schreiben dafür

z1:=3+2i

und sprechen von einer komplexen Zahl (allgemein `z=a+bi`). Hierbei heißt `a` Realteil und `b` Imaginärteil. Unter dem »Betrag« `|z|` verstehen wir die Länge des Pfeils, also in unserem Beispiel $\sqrt{13}$, allgemein (Satz des Pythagoras):

$$|z| = \sqrt{a^2+b^2} \quad \text{oder}$$
$$|z|^2 = a^2+b^2$$

Weiterhin sei noch eine zweite komplexe Zahl gegeben, sagen wir

`z2:=2+4i.`

Dann definiert man

```
z = z1+z2 = (3+2i)+(2+4i)
          = 5+6i
```

Mit dieser Definition ist `z` die Diagonale des von `z1` und `z2` aufgespannten Parallelogramms.

Entsprechend definiert man

```
z1-z2 = (3+2i)-(2+4i)
      = 1-2i
```

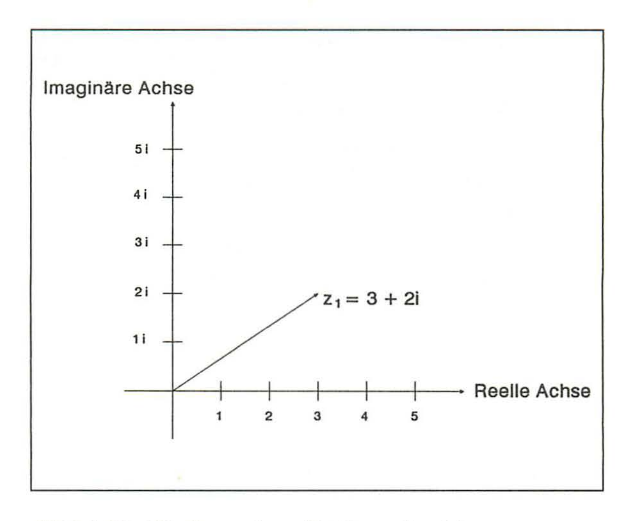


Bild 1.32: Die komplexe Zahl z = 3 + 2i

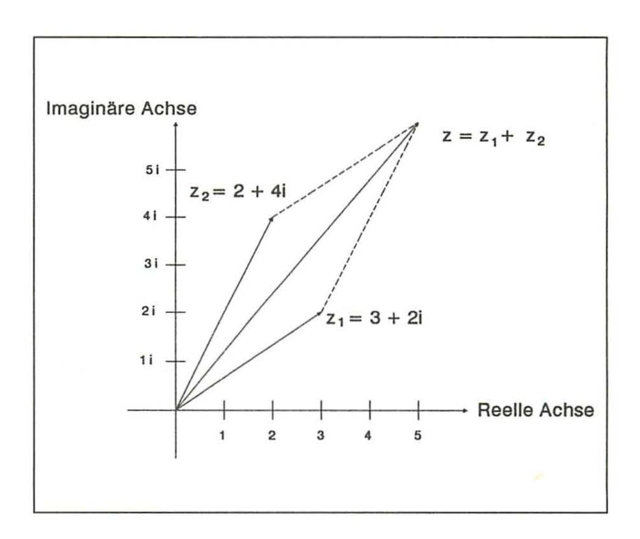


Bild 1.33: Summe zweier komplexer Zahlen

z1 schließe den Winkel α (hier 33.7°) mit der reellen Achse ein und z2 den Winkel β (hier 63.4°). Man definiert das Produkt z1*z2 als diejenige komplexe Zahl – dargestellt als Pfeil – die den Winkel α +β mit der reellen Achse einschließt und als Länge das Produkt |z1|*|z2| annimmt.

Für

```
(0+1i)(0+1i) = i*i = i²
```

ergibt sich dann -1. Allgemein gilt

```
z1*z2=(a+bi)(c+di)
     =ac+adi+bci+bdi²
     =ac+adi+bci+bd(-1)
     =(ac-bd) +(ad+bc)i
      Realteil Imaginärteil
```

In unserem Beispiel ist also

```
z1*z2=(3+2i)*(2+4i)
     =(6-8)+(12+4)i
     =2+16i
```

Für z² gilt:

```
z²=(a+bi)²=(a²-b²)+2abi
```

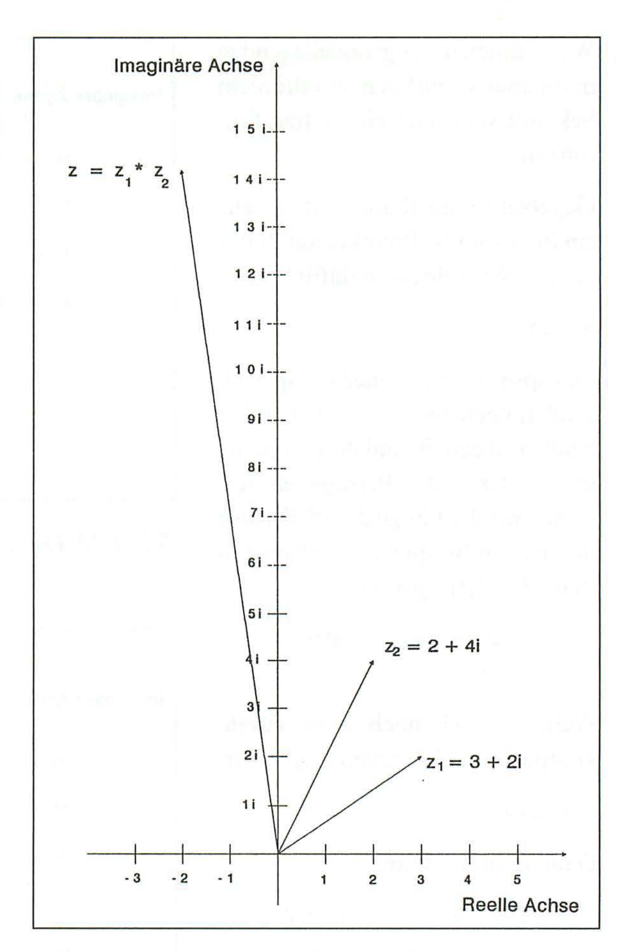


Bild 1.34: Produkt zweier komplexer Zahlen

Die Division ist nun einfach:

$$\frac{z1}{z2} = \frac{a+bi}{c+di} = \frac{(a+bi)(c-di)}{(c+di)(c-di)} = \frac{(ac+bd)+(bc-ad)i}{c^2+d^2}$$

Übertragen auf das Zahlenbeispiel ergibt sich:

$$\frac{z1}{z2} = \frac{3+2i}{2+4i} = \frac{(6+8)+(4-12)i}{4+16} = \frac{14-8i}{20} = 0.7-0.4i$$

Wir implementieren im folgenden Bibliotheksmodul `ComplexLib` die Grundrechenarten, $|z|^2$, z^2, und drei Umwandlungsfunktionen: `re`, `im` und `cmplx`:

```
DEFINITION MODULE ComplexLib;

TYPE complex = RECORD re,im : REAL END;

PROCEDURE re(z : complex) : REAL;          (* gibt den Realteil von z     *)
PROCEDURE im(z : complex) : REAL;          (* gibt den Imaginärteil von z *)
PROCEDURE cmplx(r,i : REAL) : complex;     (* gibt z :=(r,i)              *)
PROCEDURE abs2(z : complex) : REAL;        (* gibt Betrag(z)*Betrag(z)    *)

PROCEDURE addc(z1,z2 : complex) : complex;  (* gibt z1 + z2 *)
PROCEDURE subc(z1,z2 : complex) : complex;  (* gibt z1 - z2 *)
PROCEDURE mulc(z1,z2 : complex) : complex;  (* gibt z1 * z2 *)
PROCEDURE divc(z1,z2 : complex) : complex;  (* gibt z1 / z2 *)
PROCEDURE sqrc(z: complex) : complex;       (* gibt z  *  z *)

END ComplexLib.
```

Die Implementation ist einfach:

```
IMPLEMENTATION MODULE ComplexLib;

PROCEDURE re(z : complex) : REAL;
BEGIN
  RETURN z.re
END re;

PROCEDURE im(z : complex) : REAL;
BEGIN
  RETURN z.im
END im;

PROCEDURE cmplx(r,i : REAL) : complex;
VAR z : complex;
BEGIN
  z.re:=r; z.im:=i;
  RETURN z
END cmplx;

PROCEDURE abs2(z : complex) : REAL;
BEGIN
  RETURN z.re*z.re + z.im*z.im
END abs2;
```

```
PROCEDURE addc(z1,z2 : complex) : complex;
VAR z : complex;
BEGIN
  z.re:=z1.re+ z2.re;
  z.im:=z1.im+ z2.im;
  RETURN z
END addc;

PROCEDURE subc(z1,z2 : complex) : complex;
VAR z : complex;
BEGIN
  z.re:=z1.re-z2.re;
  z.im:=z1.im-z2.im;
  RETURN z
END subc;

PROCEDURE mulc(z1,z2 : complex) : complex;
VAR z : complex;
BEGIN
  z.re:=z1.re*z2.re - z1.im*z2.im;
  z.im:=z1.re*z2.im + z1.im*z2.re;
  RETURN z
END mulc;

PROCEDURE divc(z1,z2 : complex) : complex;
VAR z : complex;
    n : REAL;
BEGIN
  n:=abs2(z2);
  z.re:=(z1.re*z2.re + z1.im*z2.im)/n ;
  z.im:=(z1.im*z2.re - z1.re*z2.im)/n;
  RETURN z
END divc;

PROCEDURE sqrc(z: complex) : complex;
VAR s : complex;
BEGIN
  s.re:=z.re*z.re - z.im*z.im;
  s.im:=2.0*z.re*z.im;
  RETURN s
END sqrc;
END ComplexLib.
```

ExternerModul zur Tastaturbehandlung

Das dritte Beispiel führt Sie schon in Atari-interne Bereiche.

Die Standardprozedur `Read` kann nur die Tasten lesen, die einen ASCII-Code haben, nicht also die Cursortasten und Funktionstasten. Benutzt man die Prozedur `BConIn` aus dem Modul `BIOS`, so wird beim Drücken einer Taste ein LONGCARD-Wert zurückgegeben.

Im unteren Byte steht der ASCII-Code der Taste (falls vorhanden, sonst 0). Im dritten Byte der sogenannte ScanCode (das ist die Nummer der Taste auf der Tastatur, zum Beispiel 1 für <Esc>, 2 für die Taste <1> usw.).

Mit der Prozedur `GetKBShift` kann man zusätzlich fragen, ob eine der Metatasten <Shift>, <Control>, <Alternate> oder <Capslock> gedrückt wurde. Die Prozedur des folgenden Moduls erlaubt es, »normale« Tasten und Sondertasten zu lesen. Zu einigen oft gebrauchten Tasten werden Konstanten zur Verfügung gestellt, die in Anlehnung an die Definition des Moduls SWiSS von SPC-Modula entstanden sind. Sie erweitern sozusagen den ASCII-Code über den Bereich 0..255 hinaus. Aus diesem Grunde geben wir keinen `CHAR`-Wert, sondern eine `CARDINAL`-Zahl zurück.

```
DEFINITION MODULE Tastatur;

CONST  (* die wichtigsten Sondertasten *)
      PHoch = 256;  PTief = 257;  PLinks = 258;  PRechts = 259;

       (*  Pfeiltasten *)
      SHoch = 356;  STief = 357;  SLinks = 358;  SRechts = 359;

       (*  Shift-Pfeiltasten *)
      Help = 260;   Undo = 261;   Insert = 262;   Clear = 263;

      F1 = 283;  F2 = 284;  F3 = 285;  F4 = 286;  F5 = 287;
      F6 = 288;  F7 = 289;  F8 = 290;  F9 = 291;  F10 = 292;
       (*  Funktionstasten *)

PROCEDURE AlleTasten(VAR megataste, scan, ascii : CARDINAL);
       (*
        *  Gibt den ASCII-Wert und den Scan-Code einer Taste zurück.
        *  megataste gibt den Zustand der Shift, Alternate, Control und
        *  Capslock Taste an.
        *  0 = keine megataste gedrückt, 1 = rechte Shifttaste, 2 = linke,
        *  4 = Controltaste, 8 = Alternatetaste, 16 = Capslocktaste.
        *  Bei Kombinationen hiervon gilt die Summe.
        *)
```

```
PROCEDURE lies(VAR c : CARDINAL);
       (*
        * Liest eine Taste. Gibt den ASCII-Wert zurück oder eine der
        * oben implementierten Sondertastenwerte.
        * Für andere Tasten wird Null zurückgegeben.
        *)

END Tastatur.
```

```
IMPLEMENTATION MODULE Tastatur;

FROM SYSTEM IMPORT BYTE;

FROM BIOS   IMPORT KBShifts, Device, BConIn, GetKBShift;
PROCEDURE AlleTasten(VAR megataste, scan, ascii : CARDINAL);
VAR lc : LONGCARD;
    m  : KBShifts;
    b  : BYTE;
BEGIN
  lc := BConIn(CON);                        (* 4 Byte-Wert  *)
  ascii := SHORT(lc MOD 256L);              (* unteres Byte *)
  lc := lc DIV 10000HL;                     (* 2. Byte konstant = 0 *)
  scan := SHORT(lc MOD 256L);               (* 3. Byte *)
  m := GetKBShift();                        (* 4. Byte *)
  megataste := CARDINAL(LONG(BYTE(m)));
END AlleTasten;

PROCEDURE lies(VAR c : CARDINAL);
VAR mt,sc,as : CARDINAL;
    shift    : BOOLEAN;
BEGIN
  AlleTasten(mt,sc,as);
  shift := (mt = 1) OR (mt = 2) OR (mt = 16);
    (* rechte oder linke Shift-Taste oder Caps-Lock gedrückt? *)
  CASE sc OF
    59 : c := F1 |
    60 : c := F2 |
    61 : c := F3 |
    62 : c := F4 |
    63 : c := F5 |
    64 : c := F6 |
    65 : c := F7 |
    66 : c := F8 |
```

```
        67 : c := F9 |
        68 : c := F10|
        71 : c :=Clear|
        72 : IF shift THEN c := SHoch   ELSE c := PHoch   END |
        75 : IF shift THEN c := SLinks  ELSE c := PLinks  END |
        77 : IF shift THEN c := SRechts ELSE c := PRechts END |
        80 : IF shift THEN c := STief   ELSE c := PTief   END |
        82 : c := Insert |
        97 : c := Undo   |
        98 : c := Help   |
      ELSE c := as END;
  END lies;
  END Tastatur.
```

Externe Module können wiederum andere externe Module aufrufen. Das ist das schöne in Modula! Man kann sich auf diese Weise ganze Modulbibliotheken aufbauen. Unser Modul `Tastatur` baut auf dem Modul `BIOS` auf. Die Dienste von `Tastatur` nutzt nun ein weiterer Modul `Eingabe`, den wir im folgenden vorstellen.

`Eingabe` gibt die Möglichkeit einer komfortablen Eingabe auf dem TOS-Bildschirm. Hiermit lassen sich Eingabemasken erstellen, mit denen man Zeichenketten und Zahlen eingeben kann. Das ist zwar etwas altmodisch für den Atari, da man hier schöne »Dialogboxen« mit GEM kreieren kann, doch diese Techniken werden erst im vierten Kapitel behandelt.

Wir werden diesen Modul für eine Dateiverwaltung im Abschnitt 2.3.2. nutzen.

Das Einlesen von Zeichenketten und Zahlen ist mit unserem Modul etwas komfortabler als mit `InOut.ReadString` bzw. `ReadCard`. Megamax-Modula bietet hier aber bereits standardmäßig gute Editiermöglichkeiten.

```
DEFINITION MODULE Eingabe;

FROM DatumBib IMPORT DatumTyp;

PROCEDURE Glocke;
        (*
         *  Bewirkt Ertönen der Warnglocke.
         *)
PROCEDURE LiesZeichen(gueltig : ARRAY OF CHAR) : CHAR;
        (*
         *  Liest ein Zeichen ( 0C < Z <= 377C) von der Tastatur, das in
         *  'gueltig' enthalten ist. Andere Zeichen werden nicht akzeptiert.
         *)
```

```
PROCEDURE LiesWort(x, y        : CARDINAL;
                   laenge      : CARDINAL;
                   VAR wort    : ARRAY OF CHAR;
                   VAR EndTaste : CARDINAL);
      (*
       * Liest die Zeichenkette 'wort' von der Tastatur ein.
       * 'wort' hat maximal 'laenge' Zeichen.
       * Die übergebene Zeichenkette wird zunächst bei (x,y) auf den
       * Bildschirm geschrieben. Sie wird ggfs. auf 'laenge' gekürzt.
       * Das restliche Eingabefeld wird mit Unterstrichen markiert.
       * Die Eingabe ist mit den Tasten Linkspfeil, Rechtspfeil,
       * Backspace und Delete edierbar.
       * ClrHome löscht die bestehende Zeichenkette.
       * Die Insert-Taste schaltet zwischen Einfüge- und Überschreibemodus
       * um. Vorbelegt ist der Überschreibemodus.
       * Die Eingabe wird mit den Tasten Return, Escape und den Pfeiltasten
       * oben, unten, Shift Pfeil links und Shift Pfeil rechts beendet.
       * Diese Taste wird der Variable 'EndTaste' zur Weiterverarbeitung
       * durch das aufrufende Programm zur Verfügung gestellt.
       * Damit lassen sich einfach Eingabemasken erstellen.
       *)

PROCEDURE LiesCard(x, y        : CARDINAL;
                   min, max    : CARDINAL;
                   VAR wert    : CARDINAL;
                   VAR EndTaste : CARDINAL);
      (*
       * Liest die Kardinalzahl 'wert' ein.
       * Hierbei wird sichergestellt, daß gilt: min <= wert <= max.
       * Die Prozedur ermittelt selbst die benötigt Stellenzahl für 'wert'.
       * Die übrigen Parameter sind wie bei 'Lieswort'.
       * Die Eingabe ist wie bei 'LiesWort' edierbar.
       *)

PROCEDURE LiesDatum(x,y          : CARDINAL;
                    VAR datum    : DatumTyp;
                    VAR EndTaste : CARDINAL);
      (*
       * Liest ein Datum ein (mit Überprüfung auf Gültigkeit).
       * Schreibt das übergebene Datum an Stelle (x,y) auf den Bildschirm.
       * Das Datum ist mit den Pfeiltasten edierbar. Die Punkte im Datum
       * braucht der Anwender nicht zu tippen.
       * Die übrigen Parameter sind wie bei 'Lieswort'.      *)
END Eingabe.
```

Zunächst werden beim Einlesen soviele Unterstriche auf den Bildschirm geschrieben, wie das einzugebende Wort maximal hat (Maske). Das übergebene Wort wird angegeben. Folgende Sondertasten werden interpretiert:

- Pfeiltasten links/rechts zur Cursorsteuerung
- <Delete> zum Löschen des Zeichens, auf dem der Cursor steht
- <Backspace> zum Löschen des Zeichens links vom Cursor
- <Clr Home> zum Löschen des gesamten Wortes, der Cursor steht an der ersten Position
- <Insert> schaltet um zwischen Einfüge-Modus und dem Überschreibe-Modus. Der Modus bleibt bestehen, bis erneut die <Insert>-Taste getippt wird. Der Überschreibe- Modus ist vorbelegt.
- Diverse Tasten zur Beendigung der Eingabe: Pfeiltasten nach unten und oben, <Shift>-Pfeiltaste rechts, <Return> und <Esc>. Sie werden der Variablen `EndCh` übergeben zur Weiterverarbeitung im aufrufenden Programm.

Mit einer Pfeiltaste kann auf das nächste Eingabefeld unterhalb/oberhalb/rechts/links in einer Eingabemaske gesprungen werden. <Return> ist die Standard-Abbruchstaste und sollte einen Sprung auf das nächste Eingabefeld bewirken. <Esc> kann zum Abbruch des gesamten Dialogs genutzt werden. Doch zunächst der Implementationsmodul. Die Prozeduren sind hier etwas länger, aber trotzdem leicht verständlich, da die unterschiedlichen Eingabesteuerungen übersichtlich in CASE-Strukturen abgehandelt werden. Hier gleich ein Tip: Immer wenn sich eine längere Fallunterscheidung nach Konstanten selektieren läßt, bietet die CASE-Anweisung die eleganteste Lösung.

```
IMPLEMENTATION MODULE Eingabe;

IMPORT Strings;
IMPORT StrConv;

FROM InOut    IMPORT Write, GotoXY, WriteString, WriteCard;
FROM Tastatur IMPORT PLinks, PRechts, PHoch, PTief,
                     SLinks, SRechts, SHoch, STief ,
                     Insert, Clear, lies;
FROM DatumBib IMPORT DatumTyp, Str10, MonatLaenge, DatumZuString;

CONST ESC = 27;  RET = 13; BS = 8; DEL = 127;

PROCEDURE Glocke;
BEGIN Write(7C) END Glocke;
PROCEDURE CursorAn;
```

```
BEGIN Write(33C); Write("e") END CursorAn;

PROCEDURE CursorAus;
BEGIN Write(33C); Write("f") END CursorAus;

PROCEDURE LiesZeichen(gueltig : ARRAY OF CHAR) : CHAR;
VAR c : CARDINAL;
BEGIN
  LOOP
    lies(c);                                          (* Taste ohne Echo lesen *)
    IF (c > 255) OR (c = 0) THEN Glocke
      ELSIF Strings.Pos(CHR(c),gueltig, 0 ) = - 1 THEN Glocke
      ELSE Write(CHR(c)); RETURN(CHR(c)) END
  END
END LiesZeichen;

PROCEDURE LiesWort(x, y         : CARDINAL;
                   laenge       : CARDINAL;
                   VAR wort     : ARRAY OF CHAR;
                   VAR EndTaste : CARDINAL);
VAR
   c, pos, i               : CARDINAL;
   ok, fertig, einfuegen : BOOLEAN;
BEGIN
  Strings.Copy(wort,0,laenge,wort,ok);
  GotoXY(x,y); WriteString(wort);
  FOR i := Strings.Length(wort)+1 TO laenge DO Write("_") END; (*Markierung*)
  pos := 0;
  fertig := FALSE; einfuegen :=FALSE;
  CursorAn;
  REPEAT
    GotoXY(x+pos,y);
    lies(c);                               (* beliebige Taste ohne Echo lesen *)
    CASE c OF
      32..126,
      128..255 : IF einfuegen THEN
                   IF Strings.Length(wort) = laenge THEN Glocke ELSE
                     Strings.Insert(CHR(c),pos,wort,ok);
                     FOR i:=pos TO Strings.Length(wort)-1 DO Write(wort[i]) END;
                     INC(pos);
                   END
                 ELSE
                   IF pos = laenge THEN Glocke ELSE
                     Strings.Delete(wort,pos,1,ok);
```

```
                    Strings.Insert(CHR(c),pos,wort,ok);
                    Write(CHR(c)); INC(pos);
                  END
                END |
      PRechts   : IF pos = Strings.Length(wort) THEN Glocke ELSE INC(pos) END |
      PLinks    : IF pos = 0 THEN Glocke ELSE DEC(pos) END |
      BS        : IF pos = 0 THEN Glocke ELSE
                   DEC(pos);
                   Strings.Delete(wort,pos,1,ok);
                   Write(CHR(BS));
                   FOR i:=pos+1 TO Strings.Length(wort) DO Write(wort[i-1]) END;
                   Write("_")
                  END |
      DEL       : IF pos = Strings.Length(wort) THEN Glocke ELSE
                   Strings.Delete(wort,pos,1,ok);
                   FOR i:= pos+1 TO Strings.Length(wort) DO Write(wort[i-1]) END;
                   Write("_")
                  END |
      Insert    : einfuegen := NOT einfuegen |
      Clear     : wort[0] := 0C; pos := 0; GotoXY(x,y);
                  FOR i := 1 TO laenge DO Write("_") END |
      ESC, RET, PHoch, PTief, SLinks, SRechts, SHoch, STief
                : EndTaste := c; fertig := TRUE |
      ELSE Glocke
    END;
  UNTIL fertig;
  CursorAus;
  pos := Strings.Length(wort);
  GotoXY(x+ pos,y);
  FOR i := pos+1 TO laenge DO Write(" ") END; (* restliche Striche löschen *)
END LiesWort;

PROCEDURE LiesCard(x,y          : CARDINAL;
                   min, max     : CARDINAL;
                   VAR wert     : CARDINAL;
                   VAR EndTaste : CARDINAL);
VAR
    s, zahlwort       : Strings.String;
    stellen,pos, hilf : CARDINAL;
    gueltig           : BOOLEAN;
BEGIN
  s := StrConv.CardToStr(LONG(max),1);       (* Stellen von 'max' ermitteln *)
  stellen := Strings.Length(s);          (* = maximal benötigte Stellenzahl *)
  IF (wert < min) OR (wert > max) THEN wert := min END;
```

```
  LOOP
    zahlwort := StrConv.CardToStr(LONG(wert),1);
    LiesWort(x,y,stellen,zahlwort,EndTaste);
    pos := 0;
    hilf := StrConv.StrToCard(zahlwort,pos,gueltig);
    gueltig := gueltig & (pos = Strings.Length(zahlwort)) (* Zeichen ok ? *)
                       & (min <= hilf) & (hilf <= max);   (* Bereich ok ? *)
    IF NOT gueltig THEN Glocke ELSE EXIT END
  END;
  wert := hilf;
  GotoXY(x,y); FOR pos := 1 TO stellen DO Write(" ") END;(* Eing.feld lö. *)
  GotoXY(x,y); WriteCard(wert,stellen)       (* Zahl rechtsbündig ausgeben *)
END LiesCard;

PROCEDURE LiesDatum(x,y          : CARDINAL;
                    VAR datum    : DatumTyp;
                    VAR EndTaste : CARDINAL);
VAR
    fertig, gueltig : BOOLEAN;
    c, pos, t, m, j : CARDINAL;
    s               : Str10;
    HilfDatum       : DatumTyp;

BEGIN
  REPEAT
    DatumZuString(datum,s);
    GotoXY(x,y); WriteString(s);  (* altes Datum hinschreiben: __.__.____ *)
    fertig:=FALSE;                (* Position im Datum:  pos = 0123456789 *)
    pos := 0;
    CursorAn;
    REPEAT
      IF (pos = 2) OR (pos = 5) THEN INC(pos) END; (* Punkte überspringen *)
      GotoXY(x+ pos,y);
      lies(c);
      CASE c OF
        48..57  : IF pos < 10 THEN
                    s[pos]:=CHR(c); Write(CHR(c)); INC(pos)
                  ELSE Glocke END |
        PLinks  : IF pos = 0 THEN Glocke ELSE DEC(pos) END;
                  IF (pos=2) OR (pos=5) THEN DEC(pos) END  |
        PRechts : IF pos = 10 THEN Glocke ELSE INC(pos) END|
        ESC, RET, PHoch, PTief, SLinks, SRechts, SHoch, STief
                : EndTaste :=c; fertig := TRUE             |
      ELSE Glocke END
```

```
    UNTIL fertig;
    t:=0; FOR pos := 0 TO 1 DO t := 10*t + ORD(s[pos] )- ORD("0") END;
    m:=0; FOR pos := 3 TO 4 DO m := 10*m + ORD(s[pos] )- ORD("0") END;
    j:=0; FOR pos := 6 TO 9 DO j := 10*j + ORD(s[pos] )- ORD("0") END;
    gueltig := (0 < t ) & (t < 32) & (0 < m ) & (m <13);
    IF gueltig THEN
      WITH HilfDatum DO tag:=t; monat:=m; jahr:=j END;
      gueltig:=( t <= MonatLaenge(HilfDatum))
    END;
  UNTIL gueltig;
  datum := HilfDatum;
  CursorAus;
END LiesDatum;

END Eingabe.
```

Die Eingabe von Zahlen wurde nur für `CARDINAL`-Zahlen implementiert. Analog kann man eine Eingabe für die übrigen numerischen Typen schreiben. Mit dem folgenden kleinen Testprogramm können Sie die Einzelheiten der Eingabeprozeduren ausprobieren:

```
MODULE EingabeDemo;

FROM InOut     IMPORT WriteLn, Write, WriteString, WriteCard, WritePg, GotoXY;
FROM DatumBib IMPORT DatumTyp, Str10, DatumZuString;

IMPORT Eingabe;

CONST max =10;

VAR Wort      : ARRAY [0..max] OF CHAR;
    DatumStr : Str10;
    ende, z   : CARDINAL;
    datum     : DatumTyp;

BEGIN
  Wort:= "Hallo"; z:= 300; datum.tag:= 1; datum.monat:= 1; datum.jahr:= 1990;
  REPEAT
    WritePg;
    WriteLn; WriteString("Test des Moduls 'Eingabe'.");
    WriteLn; WriteString("Tippen Sie Worte ein, testen Sie Sondertasten!");
    Eingabe.LiesWort(2,5,max+1,Wort,ende);
    WriteString("    Eingabe: ");   WriteString(Wort);
    WriteString("    EndTaste : "); WriteCard(ende,1);
```

```
      WriteLn; WriteLn; WriteString("Tippen Sie Zahlen zwischen 0 und 1000 ein!");
      WriteLn; WriteString("Testen Sie auch Buchstaben und zu große Zahlen!");
      Eingabe.LiesCard(2,10,0,1000,z,ende);
      WriteString("     Eingabe: ");    WriteCard(z,1);
      WriteString("     EndTaste = "); WriteCard(ende,1);
      WriteLn; WriteLn; WriteString("Tippen Sie ein Datum ein!");
      WriteLn; WriteString("Testen Sie auch Buchstaben und ungültige Daten!");
      Eingabe.LiesDatum(2,15,datum,ende);
      DatumZuString(datum,DatumStr);
      GotoXY(17,15); WriteString("Eingabe: "); WriteString(DatumStr);
      WriteString("     EndTaste = "); WriteCard(ende,1);
      WriteLn; WriteLn; WriteString("Wollen Sie Weitermachen (j/n)? ");
    UNTIL CAP(Eingabe.LiesZeichen("jJnN")) = "N"
  END EingabeDemo.
```

Eine Anwendung des Moduls Eingabe zeigt die Dateiverwaltung im Kapitel (2.3.2).

In den weiteren Kapiteln werden viele solche allgemeinnützliche externe Module vorgestellt, auf diese Weise entsteht eine kleine Bibliothek.

1.7.4 Externe Standardmodule

Die Sprache Modula selbst ist mit Absicht möglichst knapp gehalten worden, schließlich können weitere Funktionen einfach aus externen Modulen importiert werden. Daher werden standardmäßig eine Reihe von externen Modulen mitgeliefert. Hierzu gehört im allgemeinen:

`InOut`
: Ein- und Ausgabe über Bildschirm und Tastatur; Umleitemöglichkeit für Drucker und Dateien

`RealInOut`
: Erweiterung von `InOut` für `REAL`-Zahlen (nur bei einigen Systemen)

`Terminal`
: Minimale, zeichenweise Ein- und Ausgabe nur für Bildschirm und Tastatur; eignet sich besonders für den TOS-Bildschirm

`Strings`
: Für Stringoperationen

`MathLib` oder `MathLib0`
: Mathematische Funktionen auf dem Typ `REAL`

`Files` oder `FileSystem`
    Ein- und Ausgabe für Dateien (Files)

`Storage`
    Speicherverwaltung für dynamische Variablen

`SYSTEM`
    ein Pseudomodul, der direkt zur Sprache gehört. Genaues siehe unten.

Bei den Atari-Systemen gibt es zusätzlich einige Module, die Schnittstellen zum Betriebssystem liefern wie `BIOS`, `XBIOS` und `GEMDOS` und die GEM-Module zu AES, VDI und Line-A-Graphik.

Darüber hinaus werden je nach Lieferumfang des Modula-Systems mehr oder weniger nützliche Module mitgeliefert, die aber im allgemeinen keinem Standard entsprechen. Hierzu gehören noch eine Reihe von Modulen zur Konvertierung von Datentypen und zu speziellen Zwecken wie zum Beispiel zum Lesen der Atari-Uhr. Wie gesagt, diese Module sind nicht genormt und leider deshalb von Modula-System zu Modula-System verschieden. Wir werden deshalb nicht großartig darauf eingehen. Zu `InOut` sind bereits viele Beispiele gegeben worden. Beispiele zu Stringbehandlung hatten wir im letzten Abschnitt. Die Dateibehandlung wird in Kapitel 2.3 besprochen. Zu `Storage` wurde schon das Wichtigste in Abschnitt 1.6.6 gesagt; siehe auch Kapitel 2.

Bleibt noch der Modul `SYSTEM`.

Der Modul SYSTEM

Der Modul `SYSTEM` gehört zu jedem Modula-System. Hier findet man systemabhängige Prozeduren und die Typen `BYTE`, `WORD`, `LONGWORD` und `ADDRESS`. Da diese Datentypen und Prozeduren rechnerabhängig sind, werden sie nicht als Bestandteil der Sprache angesehen, sondern in den »Pseudo«-Modul `SYSTEM` ausgelagert. `SYSTEM` ist eigentlich Bestandteil des Compilers; es gibt auch keinen Definitionsmodul dazu. Der Grund dafür liegt darin, daß `SYSTEM` einige Prozeduren enthält, die in Modula nicht auszudrücken sind (siehe unten `VAL`).

`SYSTEM` stellt die »Joker-Typen« `BYTE`, `WORD`, `LONGWORD` und `ADDRESS` bereit. Sie sind kompatibel zu allen anderen Typen, wenn sie in der Parameterliste von Prozeduren auftreten. Lediglich der Speicherbedarf muß mit dem der übergebenden Argumente übereinstimmen, also 1 Byte bei `BYTE`, 2 Byte bei `WORD` und 4 Byte bei `LONGWORD`.

Beispiel:

```
FROM SYSTEM IMPORT BYTE, WORD, LONGWORD;

VAR
    ch : CHAR;
    c  : CARDINAL;
    i  : INTEGER;
    bs : BITSET;
    lc : LONGCARD;
    li : LONGINT;

PROCEDURE pB (b:BYTE);
BEGIN
  <...>
END pB;

PROCEDURE pW (w:WORD);
BEGIN
  <...>
END pW;

PROCEDURE pL (l:LONGWORD);
BEGIN
  <...>
END pL;
```

Dann sind folgende Aufrufe korrekt:

```
pB(ch);
pW(c);    pW(i);    pW(bs);
pL(lc);   pL(li);
```

Wir haben diese Methode bereits im Programm `BitSetTest` aus Abschnitt 1.3.7 benutzt. Schauen Sie sich dieses Beispiel noch einmal an.

An formale Parameter vom Type `ARRAY OF BYTE`, `ARRAY OF WORD`, `ARRAY OF LONGWORD` können Daten beliebiger Länge übergeben werden; einschließlich Verbunde und Felder. Wir werden hiervon in Kapitel 2 Gebrauch machen. Mit diesen Datentypen kann man sehr flexible typunabhängige Prozeduren schreiben!

Bei manchen Modula-Implementationen haben BYTE, WORD und LONGWORD diese Joker-Funktion nicht nur in Parameterlisten, sondern sie sind zuweisungskompatibel mit jedem Datentyp der gleichen Länge. Folgendes ist dann erlaubt:

```
MODULE WordTest;

FROM SYSTEM IMPORT WORD;
VAR
    n: CARDINAL;
    b: BITSET;
    w: WORD;

BEGIN
  b := {1,3,5};
  w := b;
  n := w
END WordTest.
```

Weiterhin stellt SYSTEM den Datentyp ADDRESS bereit, der kompatibel zu jedem Pointer ist. Also

```
<...>
FROM SYSTEM IMPORT ADDRESS;

VAR
    p: POINTER TO T1;
    q: POINTER TO T1;
    a: ADDRESS

BEGIN
  a := p;
  q := a
<...>
```

ist möglich!

Folgende Prozeduren findet man in SYSTEM:

ADR(x) *address* (Adresse)

Funktion; übergibt die Adresse der Variablen x. x ist von beliebigem Typ, der Ergebnis-Typ ist ADDRESS.

`TSIZE(T)` *type-size* (»Typ-Größe«)
: Funktion mit einem Typ `T` als Argument. Ergibt den Speicherbedarf des Typs `T`. Ergebnis ist `LONGCARD`.

`VAL(T, c)` *value* (»Wert«)
: Wandelt den `CARDINAL`-Ausdruck `c` in den skalaren Typ `T` um. Damit ist die Funktion das Gegenteil der Standardfunktion `ORD`. Somit ist bei `VAR x: T;`
`ORD(VAL(T,x))= x` und
`VAL(T,ORD(x))= x`,
falls keiner der Werte zu groß für den anderen Typ ist. Damit läßt sich die Standardfunktion `CHR(c)` darstellen als `VAL(CHAR, c)`. Bei manchen Compilern ist `VAL` allerdings als universelle Konvertierungs-Funktion implementiert: `T` kann dabei jeder Typ und `c` von jedem Typ sein.

Diese drei Prozeduren enthält jeder SYSTEM-Modul. Darüber hinaus gibt es noch Prozeduren zur Behandlung von Coroutinen. Diese sind bei manchen Systemen ausgelagert; man findet sie dann in einem Modul `Coroutines`.

Das folgende Beispiel demonstriert den Umgang mit `SYSTEM`:

```
MODULE SYSTEMTest;

FROM SYSTEM IMPORT TSIZE, VAL, ADR;
FROM InOut IMPORT WriteString, WriteCard, WriteLHex, WriteLn, Write, Read;

VAR taste : CHAR;
    x,y   : REAL;

BEGIN
   WriteString("Größe von LONGCARD: "); WriteCard(TSIZE(LONGCARD), 8);
   WriteString(" Bytes");  WriteLn;
   WriteString("Typumwandlung 'A' : "); WriteCard(VAL(CARDINAL,"A"),8);
   WriteString(" als Kardinalzahl = ASCII- Wert"); WriteLn;
   WriteString("Adresse von x      : "); WriteCard(ADR(x),8);
   WriteString(" dezimal = "); WriteLHex(ADR(x),8);
   WriteString(" hexadezimal"); WriteLn;
   WriteString("Adresse von y      : "); WriteCard(ADR(y),8);
   WriteString(" dezimal = "); WriteLHex(ADR(y),8);
   WriteString(" hexadezimal"); WriteLn;
   Read(taste);
END SYSTEMTest.
```

1.7.5 Software-Engineering und Modulhierarchien

Wie wir gesehen haben, können externe Module außer Prozeduren auch Variablen, Datentypen und Konstanten bereitstellen. In unseren Beispielen kommt dies alles in Mischform vor. Man unterscheidet folgende Modulklassen als Reinformen:

1. Funktionsmodule
2. Datenmodule
3. Datenkapseln
4. Abstrakte Datentypen (ADT)

Zu 1.

Hier werden nur Prozeduren zu Verfügung gestellt (Beispiel Abschnitt 2.1; Modul `Felder`).

Zu 2.

Hier werden nur Konstanten und (oder Datentypen) exportiert. Ein Beispiel ist der Modul `GemGlobals` in Megamax-Modula oder `ASCII` von Hänisch- oder SPC-Modula.

Zu 3.

Hier wird eine Datenstruktur implementiert, der Zugriff ist aber nur über die exportierten Prozeduren möglich. Die Realisierung bleibt dem Benutzer verborgen. Ein Beispiel findet man in Abschnitt 2.2; `Stapel`.

Zu 4.

Hier wird nur der Name (!), nicht aber der Typ eines Datentyps exportiert, sowie Prozeduren, die auf dem Datentyp arbeiten. Die konkrete Realisierung des Datentyps bleibt dem Benutzer verborgen. Er braucht sich darum nicht zu kümmern. Die Typdeklaration muß natürlich im Implementationsmodul erfolgen. Beispiele sind `Schlange` und `Baum` im 2. Kapitel.

Mit eigenen externen Modulen und mitgelieferten Standardmodulen kann man übersichtlich komplexe Softwarepakete erstellen. Dies sei am Beispiel des `Eingabe`-Moduls aus dem letzten Abschnitt erläutert:

Definitions- und Implementationsmodul stützen sich auf die mitgelieferten Module `Strings`, `StrConv` und `InOut`, sowie auf unsere Module `DatumBib` und `Tastatur`. `DatumBib` stützt sich auf den Betriebssystem-Modul `GEMDOS`. `Tastatur` benötigt wiederum `SYSTEM` und `BIOS`. Der `Eingabe`-Modul wird nun seinerseits im zweiten Kapitel von einem Modul `Dateiverwaltung` benutzt, der sich wiederum auf etliche andere Module stützt, unter anderem auch auf `Tastatur`. Man erhält also folgende Hierarchie:

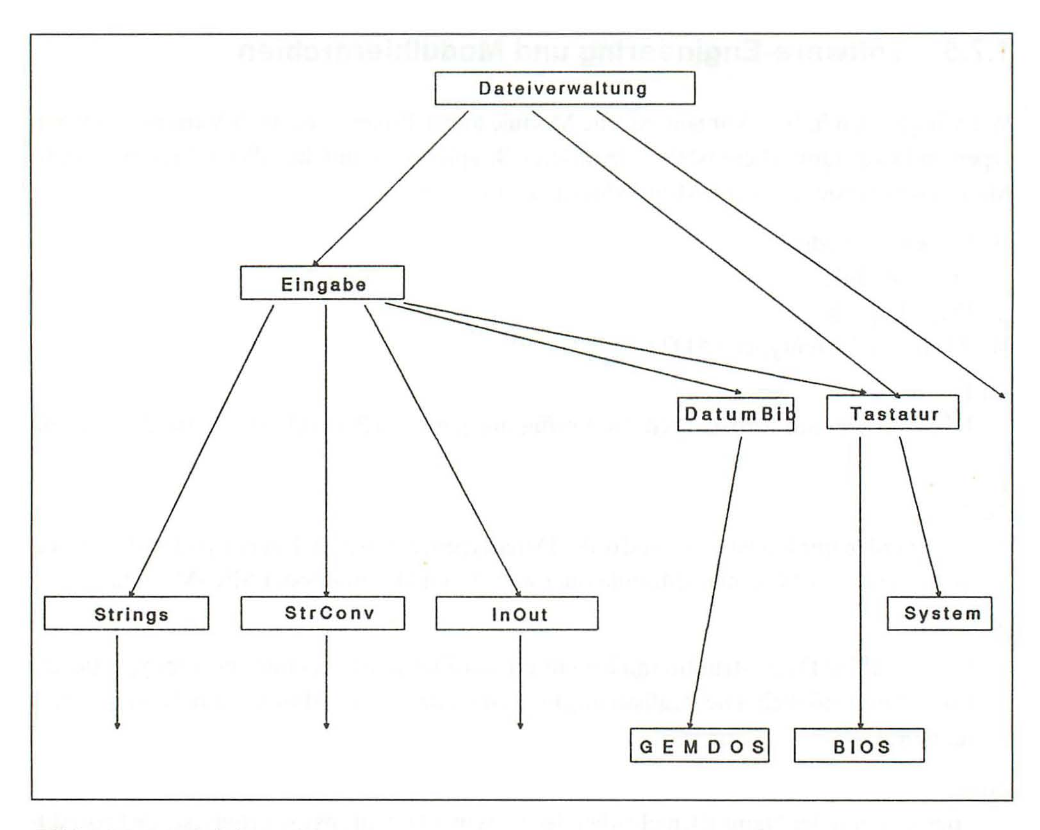


Bild 1.35: Modulhierarchien

Dieses Beispiel zeigt, daß Module sich in beliebiger Tiefe aufrufen können. Das Prinzip der hierarchischen Unterteilung von Programmen gab der Sprache Modula den Namen. Es werden damit zwei Ziele erreicht:

- Abstraktionshierarchien, wie sie bei der »Top-down«-Entwicklung von Algorithmen entstehen, können unmittelbar in eine entsprechende Modulstruktur überführt werden.
- Es bleibt alles schön übersichtlich, da man bei der Entwicklung höherer Module nur die Definitionsmodul kennen muß, die Details der Implementation sind uninteressant.

Die sich hieraus ergebenden Vorteile seien noch einmal zusammengefaßt:

- Es können Modulbibliotheken erstellt werden. Bei einem neuen Programmierprojekt braucht man nicht jedesmal »das Rad neu zu erfinden«.

- Modula-2 eignet sich hervorragend für die Erarbeitung großer Software-Pakete auch im Team.
- Durch Kapselung von compiler- oder rechnerspezifischen Details erreicht man eine hohe Portabilität der Programme.

1.8 Coroutinen und parallele Prozesse

Stellen Sie sich vor, Sie hätten einen Super-Atari mit zwei 68000-Prozessoren. Der eine Prozessor könnte eine komplizierte Berechnung durchführen, während der andere gerade eine Datei ausdruckt.

Man spricht hier von nebenläufigen (= gleichzeitigen) Prozessen. So etwas geht mit Modula, auch mit nur einem Prozessor! Jedenfalls sieht es für den Benutzer eines Programms so aus, als ob zwei Prozesse gleichzeitig abliefen. Auch für den Programmierer ist die Sichtweise so.

Der Rechner realisiert das dann allerdings intern folgendermaßen: Er arbeitet nacheinander an den jeweiligen Prozessen und wird zwischen den Prozessen dauernd hin und her geschaltet (Zeitmultiplex-Verfahren). Wenn die Steuerung der Ablaufkontrolle explizit durch einen Umschaltvorgang erfolgt, spricht man von »Coroutinen«, wenn sie nach einem »Ablauffahrplan« erfolgt, von Prozessen.

Eine Coroutine kann ihren Ablauf unterbrechen, ihren momentanen Zustand einfrieren und den Programmablauf einer anderen Coroutine überlassen. Beim Einfrieren des Zustandes werden alle Registerinhalte und die Adresse der Unterbrechungsstelle auf einem gesonderten Speicherbereich »gerettet«. Wenn die Coroutine wieder an der Reihe ist, arbeitet sie so, als wäre zwischendurch nichts geschehen. Also bleiben auch Werte lokaler Variablen erhalten!

In Modula kann man aus Prozeduren Coroutinen machen. Dies funktioniert nur mit parameterlosen und nicht-rekursiven Prozeduren. Diese Prozeduren dürfen auch nicht Unterprozeduren von anderen Prozeduren sein.

Die Kommunikation zwischen den Coroutinen wird also nur durch globale Variablen abgewickelt. Das tut der Sache aber keinen Abbruch. Um die Sichtbarkeitsbereiche von Variablen begrenzt zu halten, kann man einen lokalen Modul benutzen.

Prozeduren für parallele Prozesse

Die benötigten Prozeduren für dieses Konzept sind `NEWPROCESS` und `TRANSFER`; bei manchen Systemen kommt noch `IOTRANSFER`, `IOCALL` und `LISTEN` hinzu. Man findet sie üblicherweise im Pseudomodul `SYSTEM` oder in einem separaten Modul `Coroutines`.

Wie macht man nun aus einer Prozedur eine Coroutine? Hierzu dient die Prozedur

```
PROCEDURE NEWPROCESS(p: PROC;
                     Arbeitsspeicher: ADDRESS;
                     ArbeitsspeicherGroesse: CARDINAL;   (* Megamax: LONGCARD *)
                     VAR neuerProzess: ADDRESS);
```

Hiermit wird für die parameterlose Prozedur (Datentyp: PROC, vgl. 1.6.7) ein Prozeß erzeugt. Dieser Prozeß benötigt einen Arbeitsspeicher, den man mit übergeben muß. Als Ergebnis erhält man in neuerProzess die Coroutinen-Adresse des Prozesses.

Beispiel:

```
PROCEDURE TuWas;
BEGIN <...> END TuWas;

VAR ArbeitsSpeicher: ADDRESS;
    TuWasProzess   : ADDRESS;

<...>
Storage.ALLOCATE(ArbeitsSpeicher, 1000D);
NEWPROCESS(TuWas, ArbeitsSpeicher, 1000D, TuWasProzess);
```

Hierdurch wird aus der Prozedur TuWas die Coroutine TuWasProzess. Den benötigten Speicherplatz holen wir uns hier mittels ALLOCATE vom Heap. Wir hätten auch einfach die Adresse eines 1000-Byte großen Feldes (VAR feld: ARRAY[0..999] OF CHAR) übergeben können:

```
NEWPROCESS(TuWas,ADR(feld), 1000L, TuWasProzess);
```

Die Coroutine haben wir nun, aber ihr Zustand ist noch »schlafend«. Mit

```
TRANSFER(VAR quelle, ziel: ADDRESS);
```

weckt man sie und friert gleichzeitig die aufrufende Coroutine oder das aufrufende Programm ein. Die Adresse der aufrufenden Coroutine wir in quelle abgelegt, hier kann durch einen erneuten Aufruf von TRANSFER – jetzt mit vertauschten Parametern – mit der Arbeit fortgefahren werden. Aufrufe von TRANSFER ermöglichen also die Weiterarbeit am letzten Unterbrechungspunkt.

Hier laufen drei Routinen parallel:

1. Eine reelle Zahl wird laufend um 1 erhöht, quadriert und ausgedruckt.

2. Auf den Drucker werden fortlaufend Zeichen ausgegeben.

3. Die Tastatur wird dauernd abgefragt. Eventuell eingegebene Zeichen werden auf den Bildschirm geschrieben.

Die folgende Demonstration zeigt den Umgang mit Coroutinen.

```
MODULE CoroutinenDemo;

FROM SYSTEM   IMPORT NEWPROCESS, TRANSFER, ADDRESS, ADR, BYTE;
FROM GEMDOS   IMPORT PrnOS;
FROM InOut    IMPORT GotoXY, WriteReal, Write, WriteString, BusyRead;
FROM Files    IMPORT File, Open, Close, Access;
IMPORT Text;

VAR    TransAdr : RECORD
                    HauptPrg,
                    quadrat, druck, taste : ADDRESS
                  END;
     pr         : File;

PROCEDURE quadrat;
VAR x : REAL;
BEGIN
  x := 1.0;
  LOOP
    x:= x + 1.0;
    GotoXY(10,10);
    WriteReal(x,4,2);
    WriteString(" -> ");
    WriteReal(x*x,10,2);
    TRANSFER(TransAdr.quadrat,TransAdr.druck)
  END
END quadrat;

PROCEDURE druck;        (*Dank Coroutinekonzept funktioniert diese     *)
VAR                     (*Prozedur sogar ohne Drucker! Sie gibt ohne   *)
  ch : CHAR;            (*Drucker sofort wieder die Kontrolle weiter   *)
BEGIN
  ch := " ";
  LOOP
    IF PrnOS() THEN                (*Wenn der Drucker bereit ist      *)
      Text.Write(pr,ch);
      IF ch = CHR(110) THEN ch := " ";Text.Writeln(pr)  ELSE INC(ch) END;
```

```
  END;
    TRANSFER(TransAdr.druck,TransAdr.taste)
  END;
END druck;

PROCEDURE taste;
VAR key : CHAR;
BEGIN
  LOOP
    BusyRead(key);                (*Wenn eine Taste gedrückt wurde    *)
    IF key # 0C THEN
      GotoXY(10,15); Write(key);
      IF key  = 33C THEN TRANSFER(TransAdr.taste,TransAdr.HauptPrg) END
    END;
    TRANSFER(TransAdr.taste, TransAdr.wurzel)
  END
END taste;

VAR Platz :   RECORD
                quadrat, druck, taste : ARRAY [0..1999] OF BYTE
              END;

BEGIN
  Open(pr,"PRN:",writeSeqTxt);
  NEWPROCESS(wurzel,ADR(Platz.quadrat),SIZE(Platz.quadrat),TransAdr.quadrat);
  NEWPROCESS(druck,ADR(Platz.druck),SIZE(Platz.druck),TransAdr.druck );
  NEWPROCESS(taste,ADR(Platz.taste),SIZE(Platz.taste),TransAdr.taste );

  GotoXY(20,2); WriteString("Coroutinen Demo, Abbruch mit ESC");
  GotoXY(20,4); WriteString("Bitte den Drucker einschalten!");
  GotoXY(20,5); WriteString("Es laufen 3 Prozesse parallel:");
                (*Wenn eine Taste gedrückt wurde    *)
  GotoXY(20,6); WriteString("Berechnung, Tastaturabfrage mit Echo Druck.");

  TRANSFER(TransAdr.HauptPrg,TransAdr.taste);

  Close(pr);
END CoroutinenDemo.
```

Bei diesem Programm haben wir Endlos-Schleifen eingesetzt. Eine Coroutine endet nämlich nicht, sondern sie gibt ihre Kontrolle ab. Das Ende einer Coroutine bedeutet das Ende des Programms. Man erkennt an diesem Beispiel weiter, daß das Hauptprogramm auch zur Coroutine wird.

Vielleicht fragen Sie sich nach diesem Beispiel, wozu überhaupt das Coroutinen-Konzept gut sein soll. Ohne dieses hätte man das Beispielprogramm auch schreiben können. Da nur ein Prozessor vorhanden ist, wird ja sowieso alles nacheinander abgearbeitet.

Die Antwort: Sicherlich läßt sich auch ohne die Parallel-Programmierung auskommen. Sie ist aber in vielen Fällen praktisch, da der Progammierer die einzelnen Coroutinen unabhängig voneinander programmieren kann. Die Umschaltung überläßt er dem System. Ebenso gut könnte man ja auch ohne Verwendung von Prozeduren programmieren, aber wer will diesen Komfort schon missen?

Typische Einsatzgebiete von Coroutinen sind locker gekoppelte Vorgänge, bei denen nur wenige Daten übertragen werden müssen. Ein Beispiel wäre die Programmierung eines Weltraumspiels, bei dem mehrere Objekte mit wenigen Wechselwirkungen umherfliegen. Für jedes Objekt schreibt man hier eine Coroutine. Ein seriöses Beispiel ist der Kompilierungsvorgang. Hier sind das Scannen, Parsen und die Codeerzeugung relativ lose gebunden.

1.9 Hinweise zum guten Programmierstil in Modula-2

Hierzu ist an verschiedenen Stellen bereits etwas gesagt worden. Das folgende kann als Richtschnur dienen. Wir halten uns jedoch auch nicht immer sklavisch daran.

Die Verwendung von Konstanten

1. Konstanten machen ein Programm übersichtlicher und portierbarer

2. Besser als

```
VAR feld: ARRAY [0..999] OF T;
```

ist

```
CONST
    max = 999;
VAR
    feld: ARRAY[0..max] OF T;
```

3. Die zweite Version ist wesentlich leichter abzuändern (Aspekt der Wartbarkeit von Programmen). Es ist einfacher, das Feld bei Bedarf größer zu machen, so daß Konstrukte wie

   ```
   FOR i := 0 TO max DO  feld[i]:=0 END;
   ```

   immer noch wie gewünscht arbeiten.

Die Verwendung von Prozeduren

Prozeduren sollten so bezeichnet werden, daß aus ihnen hervorgeht, was sie tun. Umgekehrt sollten sie auch nicht mehr machen, als ihr Bezeichner aussagt.

Für Funktionen verwendet man vorzugsweise Substantive, die die Werte beschreiben, die sie liefern. Bei Booleschen Funktionen benutzt man auch Adjektive (zum Beispiel `leer`) oder Partizipien (`gefunden`). Alle übrigen Prozeduren tun etwas, daher verwendet man Verben (meist Infinitiv (`Einrichten`) oder Imperativ (`Schreibe`)).

Eine Prozedur sollte nicht allzu lang sein, sonst verliert man den Überblick. Ein guter Richtwert sind die 25 Zeilen des Bildschirms. Einzeiler sind erlaubt.

Prozeduren in großen Programmen sollten möglichst nur über die klar ersichtliche Schnittstelle Parameterliste kommunizieren und nicht mit globalen Variablen operieren, da dies zu unerwünschten Seiteneffekten führen kann. Auf jeden Fall sollten die Laufvariablen stets lokal sein.

Kommen in der Parameterliste einer Prozedur mehrere Variablen vor, so setzt man Eingabevariablen nach vorne und Ausgabevariablen nach hinten. Variablen, die sowohl eine Eingabe liefern als auch modifiziert werden, kommen dazwischen.

```
PROCEDURE Kopiere(quelle: Typ; VAR ziel: Typ);
```

Insgesamt sollte die Anzahl der Variablen einer Prozedur nicht allzu hoch sein. Faustregel: Über fünf wird es unübersichtlich.

Externe Module

Größere Teilaufgaben, vor allem wenn sie öfter vorkommen, gehören in einen externen Modul. Dies erspart auch Zeit bei der Entwicklung von Programmen, da man separat kompilieren und testen kann.

Jeder Modul sollte nur einen beschränkten Satz an Prozeduren enthalten, die sich einer gemeinsamen Aufgabenstellung überordnen lassen. Keinesfalls sollte hier ein Wirrwarr von verschiedenen Aufgabenstellungen erledigt werden (»Können wir diese Prozedur noch brauchen?« – »Nee, aber packen wir sie mal in den Modul...«).

Der Definitionsmodul sollte deutlich machen, was die Prozeduren leisten. In den meisten Fällen ist hier eine zusätzliche Kommentierung hilfreich. Kommentare über die Funktionsweise gehören allerdings nur in das Implementationsmodul. Der Definitionsmodul sollte auch möglichst knapp gehalten werden.

Man sollte sich im Einzelfall überlegen, ob man einen Import in der Version ohne `FROM` nicht einem Import mit `FROM` vorzieht. Im ersten Fall sind die Bezeichner dieses Moduls bei der Verwendung qualifiziert zu verwenden. Dies führt zwar zu mehr »Tipparbeit«, falls der Bezeichner öfters verwendet wird, hat aber den Vorteil, daß man im Programmtext sofort sieht, aus welchem Modul er stammt; der Name ist so aussagekräftiger.

Beim Import aus Standardmodulen ist allerdings der Import mit `FORM` üblich.

Die äußere Form von Programmen

Hierzu gehören vernünftige Kommentare und Einrückungen. Kommentare dienen sicherlich dazu, ein Programm lesbarer zu machen, wenn man an ihnen schnell den Zweck der kommentierten Stelle erkennen kann, ohne sich mühevoll durch Prozeduraufrufe und mysteriöse Pointerzuweisungen quälen zu müssen. Unsinnig sind allerdings Kommentare wie

```
a := 3; (*a wird 3 zugewiesen*)
```

An dieser Stelle ein kleiner Trick mit Kommentarklammern: Man kann sie in der Testphase auch einsetzen, um Programmpassagen für den Compiler auszublenden. Hierbei können Kommentare selbst wieder von Kommentarklammern eingeschlossen werden. Das macht nichts, denn Modula unterstützt geschachtelte Kommentare:

```
(*Dies ist ein (*geschachtelter*) Kommentar*)
```

Für Einrückungen gilt nur eine Regel: Sinnvoll ist das, was übersichtlich ist. Dummerweise hat davon jeder seine eigene Vorstellung, deshalb läßt sich schwer ein Standard festlegen.

Auf jeden Fall wird das Programm unübersichtlich, wenn man Zeilenumbrüche nur macht, weil die Zeile gerade zu Ende ist. Man kann aber durchaus einmal zwei Anweisungen in eine Zeile setzen:

```
WriteLn; WriteString("Ergebnis"); WriteReal(x,5);
```

Und

```
FOR i := 0 TO max DO feld[i] = 0 END;
```

ist oft besser als

```
FOR i := 0 TO max
  DO
    feld[i] := 0
  END;
```

Sicherlich sollten Schachtelungen eingerückt werden. Professionelle Programmierer nehmen oft einen ganzen Tabulatorschritt (TAB = 8 Leerzeichen) was etwas reichlich ist; andererseits ist ein einziges Leerzeichen pro Ebene zu wenig, weil man dann schon ein Lineal anlegen muß, um festzustellen, was zusammengehört.

Wir hoffen, mit unserer Formatierung eine Version gebracht zu haben, an der man sich orientieren kann.

KAPITEL 2

Die Behandlung von Datenstrukturen in Modula

Eine Standardaufgabe der Datenverarbeitung ist das Abspeichern großer Datenmengen und anschließend das schnelle Wiederfinden bestimmter Daten. Die Daten lassen sich in Modula-2 mittels der Datenstrukturen Feld, Liste oder Baum im Speicher unterbringen; darüber hinaus besteht die Möglichkeit, die Daten (mit Hilfe dieser Strukturen) auf Dateien zu schreiben. Die Verwendung von verzeigerten Strukturen wird im Kapitel 2.2 besprochen, die von Dateien im Kapitel 2.3; zunächst geht es um Felder.

2.1 Die Behandlung von Feldern

Ziel dieses Abschnitts ist es, einen Modul zur Behandlung von Feldern beliebigen Datentyps (Suche und Sortierung) zu geben.

Als Beispiel für einen solchen Datentyp möge der folgende Verbund `KundenTyp` dienen:

```
TYPE
    str20 = ARRAY [0..19] OF CHAR;
    str30 = ARRAY [0..29] OF CHAR;
    KundenTyp = RECORD
                  KundenNr       : CARDINAL;
                  Name, Vorname  : str20;
                  Strasse        : str30;
                  PLZ            : [1000..9999];
                  Ort            : str20
                END;
```

mit dem Feld

```
VAR feld : ARRAY [0..99] OF KundenTyp;
```

Sind in einem solchen Feld nun alle 100 Kunden mit ihrer Anschrift eingelesen, so kommt es vor, daß man (zum Beispiel innerhalb eines Fakturierungs-Programmes) zu einer Kundennummer die entsprechende Kundenanschrift wissen will; das Feld muß nach der Kundennummer durchsucht werden. Die Kundennummer bildet den »Schlüssel« für den Zugriff auf die benötigten Daten, die man unter dem entsprechenden Feldindex `feld[i]` findet. Abstrahierend können wir also von einem Feld der Form

```
CONST max = 99;
TYPE verbund = RECORD
                 schluessel  : CARDINAL;
                 information : InfoTyp
```

```
            END;
VAR feld : ARRAY [0..max] OF verbund;
```

ausgehen.

Sequentielles Suchen in einem Feld

Bezeichnen wir mit `gesucht` einen vorgegebenen Schlüssel (hier eine bestimmte Kundennummer), so ist das Feld nach dem Index `i` zu durchsuchen, für den gilt:

```
gesucht = feld[i].schluessel.
```

Wir formulieren dies sogleich in einer Prozedur, wobei das gesamte Feld von Anfang an durchlaufen wird, solange bis der gesuchte Schlüssel gefunden ist. Diesen Vorgang nennt man »sequentielles Suchen«. Die Funktionsprozedur gibt den gesuchten Index zurück, mit dem dann auf den gesamten Verbund (mit Name, Anschrift etc.) zugegriffen werden kann. Ist der Schlüssel nicht vorhanden, wird `max+1` zurückgegeben, damit das aufrufende Programm erkennt, daß die Suche fehlgeschlagen ist.

```
PROCEDURE SequentielleSuche(gesucht: CARDINAL): CARDINAL;
VAR i: CARDINAL;
BEGIN
  FOR i := 0 TO max DO
    IF feld[i].schluessel = gesucht THEN RETURN i END;
  END;
  RETURN max + 1
END SequentielleSuche;
```

Sobald also der gesuchte Schlüssel gefunden ist, wird der entsprechende Index `i` mittels `RETURN` als Ergebnis zurückgegeben und die Prozedur (einschließlich der `FOR`-Schleife) abgebrochen. Nur wenn der Schlüssel nicht vorhanden ist, wird die `FOR`-Schleife vollständig durchlaufen und `max+1` zurückgegeben.

Schlüssel sind aber nicht immer `CARDINAL`-Zahlen, sondern es sind zum Beispiel bei Namen, Bestellnummern usw. oft Zeichenketten üblich. Dann läßt sich die Gleichheitsabfrage in der 5. Zeile nicht einfach mit dem Gleichheitszeichen durchführen. So importiert man – um Zeichenketten vergleichen zu können – die Prozedur `Compare` und den Typ `Relation` aus dem Modul `Strings`. Die Abfrage lautet dann:

```
IF Compare(feld[i].schluessel,gesucht) = equal THEN RETURN i END;
```

Bei dem Algorithmus »sequentielle Suche« ist mindestens eine Gleichheitsabfrage nötig, maximal max+1. Wenn wir davon ausgehen, daß alle Schlüssel gleichwahrscheinlich sind, benötigt man im Mittel max/2+1Vergleiche. Bei sehr großen Feldern (größenordnungsmäßig max>100) ist dieses einfache Verfahren vor allem bei häufiger Suche zu langsam. Dann ist es sinnvoll, das Feld zunächst nach Schlüsseln aufsteigend zu sortieren und dann mittels »binärem Suchen« zu durchsuchen, was im folgenden beschrieben werden soll.

Binäres Suchen in einem geordnetem Feld

Als Beispiel greifen wir der Einfachheit halber wieder auf ein Feld mit CARDINAL-Schlüsseln zurück. Die Daten im Feld seien so abgespeichert, daß gilt:

feld[i].schluessel<feld[j].schluessel für alle i<j.

Im Klartext: das Feld ist nach Schlüsseln aufsteigend sortiert, so daß links die Elemente mit den »kleinen« Schlüsseln, rechts die mit den »großen« stehen.

Nun können wir beim Suchen nach einem Schlüssel folgendermaßen vorgehen: Wir testen zunächst, ob die Feldkomponente mit dem »mittlerem Index« feld[mitte].schluessel der gesuchte Schlüssel ist; wobei wir mitte:=(links+rechts) DIV 2 setzen. Es gibt dann drei Fälle:

1. Der Schlüssel ist gefunden.
2. Es gilt feld[mitte].schluessel<gesucht, dann kann sich der gesuchte Schlüssel nur im rechtem Teilfeld befinden. Wir durchsuchen nun das Feld vom Index mitte+1 an, das heißt, die Suche wird mit links:=mitte+1 wiederholt.
3. Es gilt feld[mitte].schluessel>gesucht. Man hat dann im linken Teilfeld weiterzusuchen, folglich wird rechts:=mitte-1 gesetzt und mit der Suche fortgefahren.

Insgesamt ergibt sich damit als erste Version die Prozedur:

```
(* schlechte Version! *)
PROCEDURE BinaereSuche(gesucht: CARDINAL): CARDINAL;
VAR links,rechts,mitte: CARDINAL;
    gefunden:           BOOLEAN;

BEGIN
  links := 0; rechts := max;
  gefunden := FALSE;
  REPEAT
    mitte := (links + rechts) DIV 2;
    IF feld[mitte].schluessel = gesucht THEN gefunden := TRUE END;
    IF feld[mitte].schluessel < gesucht
```

```
      THEN links := mitte + 1
      ELSE rechts := mitte - 1 END
  UNTIL gefunden OR (links>rechts);
  IF gefunden THEN RETURN mitte ELSE RETURN max+ 1 END
END BinaereSuche;
```

Diese Prozedur läßt sich aber noch verbessern. Die Wahrscheinlichkeit, direkt das gesuchte Element zu finden, beträgt 1/n (n = Anzahl der Feldelemente). Es ist somit unwahrscheinlich, daß die Schleife frühzeitig abbricht. Die Abbruchsbedingung der REPEAT-Schleife ist kompliziert und innerhalb der Schleife befinden sich gleich zwei IF-Abfragen. Trotzdem wird diese Prozedur in den meisten Lehrbüchern so formuliert. Wir machen es kürzer: Wir prüfen zunächst, ob das mittlere Feldelement `feld[mitte]` einen kleineren Schlüssel hat als `gesucht`. Wenn ja, suchen wir im Teilfeld `feld[mitte+1]` bis `feld[max]` weiter, ansonsten in der anderen Hälfte `feld[0]` bis `feld[mitte]`. So fahren wir mit der Halbierung des Feldes fort, bis es nur noch aus einer Komponente besteht. Entweder ist dies die gesuchte oder aber der Schlüssel ist nicht vorhanden und es wird `max+1` als Fehlanzeige zurückgegeben.

```
PROCEDURE BinaereSuche(gesucht: CARDINAL): CARDINAL; (* schnellere Version *)
VAR links, rechts, mitte: CARDINAL;

BEGIN
  links := 0; rechts := max;
  WHILE links < rechts DO
    mitte := (links + rechts) DIV 2;
    IF feld[mitte].schluessel < gesucht THEN links := mitte+1
    ELSE rechts := mitte END
  END;
  IF feld [links].schluessel = gesucht THEN RETURN links  (* gefunden *)
 ELSE RETURN max + 1 END                              (* nichts gefunden *)
END BinaereSuche;
```

Auf der Diskette finden sie das Programm `SuchDemo`, mit dem Sie im einzelnen sehen können, wie beide Verfahren arbeiten.

Für String-Schlüssel importiert man aus dem Modul Strings die Prozedur `Compare` und den Typ `Relation`. Die 7. und die 10. Zeile müssen dann so lauten:

```
IF Compare(feld[i].schluessel,gesucht) = less THEN <...>
IF Compare(feld[i].schluessel,gesucht) = equal THEN <...>
```

Der Vorteil des Verfahrens liegt auf der Hand: hat man zum Beispiel n = 1024 Feldelemente, so erfolgen innerhalb der Schleife nur 10 Vergleiche, da das zu durchsuchende Feld jedesmal hal-

biert wird. Hinzu kommt noch die Probe auf Gleichheit nach Abschluß der Schleife. Insgesamt sind also 11 Vergleichsoperationen notwendig, unabhängig davon, ob das Element gefunden wurde oder nicht. Bei 1025 Elementen sind es 11 bis 12 Vergleiche, also ist `log2(n)+1` eine obere Schranke für die Anzahl der Vergleiche. Bei der sequentiellen Suche würde man für die gleiche Feldgröße durchschnittlich 512 Vergleiche benötigen; das entspricht etwa einem Faktor von 50! Dieser Faktor (sequentielles Suchen/binäres Suchen) wächst mit der Zahl der Feldelemente.

Nicht verschwiegen werden soll dabei der (zeitliche) Aufwand für das Sortieren des Feldes. Bezeichnen wir diesen mit `F`, den Aufwand für das binäre Suchen mit `B` und den für das sequentielle Suchen mit `S`, so lohnt sich das Sortieren des Feldes wenn gilt

```
m * S > m * B + F
```

wobei *m* die Zahl der zu erwartenden Suchvorgänge ist. Diese Ungleichung ist jedoch recht oft erfüllt. Man schätzt, daß über ein Viertel der gesamten Rechenzeit aller Computer für Sortiervorgänge verwendet wird.

Wir wenden uns im folgenden dem Sortieren von Feldern zu und beschreiben zwei Algorithmen; zunächst einen einfachen (und langsamen), dann einen komplizierteren (und schnellen). Ein Sortieralgorithmus soll aus einem unsortierten Feld ein sortiertes erstellen. Ein Feld sei sortiert, wenn gilt:

```
feld[i].schluessel≤feld[j].schluessel für alle i<j.
```

Einfaches Sortieren durch Austauschen

Das Verfahren ist ganz einfach: man durchsucht das Feld nach dem kleinsten Element und bringt es durch Vertauschen an die erste Stelle (mit Index 0). An die zweite Stelle soll das zweitkleinste Element; also durchsuchen wir das restliche Feld (vom Index 1 an) nach dem kleinsten Element und bringen es dorthin. So fahren wir fort, bis wir alle Stellen richtig belegt haben. Das realisieren wir mit zwei `FOR`-Schleifen (die äußere bestimmt die Stelle für das jeweilig kleinste Element, welches mit der inneren gesucht wird):

```
PROCEDURE sortiere;
VAR i,j,minimum: CARDINAL;
BEGIN
  FOR i := 0 TO max-1 DO
    minimum := i;
    FOR j := i + 1 TO max DO
      IF feld[j].schluessel < feld[minimum].schluessel
          THEN minimum := j END
    END;
```

```
    vertausche(feld[i],feld[minimum])
  END
END sortiere;
```

Es bleibt nur noch die Prozedur `vertausche` zu erklären. Man vertauscht zwei Variablen einfach mittels Zuweisungen über einen Zwischenspeicher `hilf`:

```
PROCEDURE vertausche (VAR a,b: verbund);
VAR hilf: verbund;
BEGIN
  hilf := a; a := b; b := hilf
END vertausche;
```

Statt des Prozeduraufrufs von `vertausche` in der Prozedur `sortiere` läßt sich selbstverständlich das Vertauschen auch explizit formulieren:

```
hilf := feld[i]; feld[i] := feld[minimum]; feld[minimum] := hilf;
```

`hilf` ist dann in `sortiere` zu deklarieren. Durch den entfallenen Prozeduraufruf erreicht man eine höhere Geschwindigkeit.

Falls der Verbund aber sehr groß ist (unser eingangs erwähnter Kundentyp belegt 104 Byte), ist der obige Austausch etwas langsam, da bei der Ausführung insgesamt 3 x 104 Byte (= 312 Byte) zugewiesen werden. In diesem Falle empfiehlt es sich, das Austauschen der Byte von einer Prozedur auf Assembler-Ebene ohne Benutzung eines Hilfsspeichers vornehmen zu lassen. Wie man so was schreibt, zeigen wir ausführlich im Kapitel 3. In Megamax-Modula ist eine solche Prozedur unter dem Namen `SwapVar` im Modul `SysUtil0` schon vorhanden; der Megamax-Programmierer schreibt also in der 10. Zeile:

```
SwapVar(feld[i],feld[minimum]);
```

Wenn das zu sortierende Feld sehr viele Elemente hat, wird das obige Sortierverfahren sehr langsam. Die innere Schleife wird von der äußeren `max`-mal aufgerufen mit jeweils 1 bis `max` Vergleichen. Das macht insgesamt

$$\text{max} + (\text{max}-1) + (\text{max}-2) + \ldots + 2 + 1 = \frac{\text{max}}{2}(\text{max}+1)$$

Vergleiche. Die Anzahl der Vergleiche wächst also quadratisch mit der Anzahl der Feldkomponenten; man sagt, der Aufwand ist von der Ordnung $O(n^2)$ (großer Buchstabe »O«, keine Null!). Das ist (leider) bei allen einfachen Sortierverfahren der Fall. Darum führen wir hier nicht weitere mehr oder weniger triviale Sortieralgorithmen auf (wie in vielen Informatikbüchern als Seitenfüller üblich), sondern erklären gleich den stets bei großen Feldern angewandten Algorithmus »Quicksort«. Quicksort ist ein auf *C. A. R. Hoare* zurückgehendes Verfahren mit einem Aufwand der Ordnung $O(n*\log_2(n))$.

Schnelles Sortieren eines Feldes (Quicksort)

Zur Erläuterung ist es hilfreich, sich vorzustellen, wie man selbst größere »Datenmengen« (z.B. in Form von Karteikarten) sortieren würde. Nehmen wir mal an, wir hätten einen Stapel mit rund 1000 Adreßkarten (unsere Kundenkartei) vor uns und kämen auf die Idee, diese zu sortieren. Nun, 1000 Karten sind sicherlich zu viel, als daß man sie mal eben »aus der Hand« sortieren kann. Also nimmt man erst mal eine »Grobsortierung« vor: man verteilt die Karten auf mehrere Gruppen (A–E, F–J, ..., oder einfach nach Anfangsbuchstaben). Diese kann man dann getrennt sortieren (falls diese dazu immer noch zu groß sind, kann man sie – jede für sich – wieder »grobsortieren« usw. Spätestens wenn man nur noch zwei Karten in der Hand hat, dürfte es keine Probleme mehr geben...). Danach muß man die Grüppchen nur noch zusammenlegen und der Stapel ist sortiert.

Genauso arbeitet auch Quicksort. Hier wird das Feld bezüglich eines mittleren Elementes x in zwei Teilfelder zerlegt, so daß die Elemente im ersten Teilfeld kleiner als x sind, die im anderen größer. Als x nimmt man irgendein Element aus dem (noch ungeteilten) Feld, im einfachsten Fall das erste. Das »Zerlegen« eines Teilfeldes geschieht durch Umverteilen der Elemente; dazu sucht man von links das erste Element, das größer ist als x und von rechts das erste kleinere (dies geschieht in der Prozedur in den beiden innersten WHILE-Schleifen) und tauscht dann die beiden Elemente. Das wiederholt man, bis links alle kleineren und rechts alle größeren als x stehen (die beiden Hälften können ungleich groß sein, schlimmstenfalls besteht eine Hälfte nur aus einem Element – dem x). Die beiden kleineren Teile werden dann wieder getrennt sortiert – das geschieht, indem Quicksort sich einfach rekursiv aufruft – bis ein Teilfeld nur noch aus einem Element besteht. Dieses ist dann zwangsläufig sortiert:

```
PROCEDURE QuickSort(links,rechts: CARDINAL);
VAR a,b: CARDINAL;
    mittlerer: CARDINAL;

BEGIN
  a := links; b := rechts;
  mittlerer := feld[(a+b) DIV 2].schluessel;
  REPEAT
  WHILE feld[a].schluessel < mittlerer DO INC(a) END;
  WHILE feld[b].schluessel > mittlerer DO DEC(b) END;
  IF a <= b THEN
    vertausche(feld[a],feld[b]);
    INC(a); DEC(b)
  END;
  UNTIL a > b;
  IF links < b THEN QuickSort(links, b) END;
  IF a < rechts THEN QuickSort(a,rechts) END;
END QuickSort;
```

Als Vergleichselement für das Zerlegen wird hier das Element in der Feldmitte ausgewählt. Wenn dies zufällig auch größenmäßig zumeist das mittlere Element ist, so zeigt die Theorie, daß $n*\log_2(n)$-Vergleiche und $n*\log_2(n)/6$-Austauschoperationen nötig sind. Das Schlimmste was passieren kann, ist, daß das »mittlere Element« immer das größte (oder kleinste) Element des Teilfeldes ist. Dann besteht das eine Teilfeld aus einem Element, das andere aus $n-1$ Elementen. Damit wird der Aufwand wieder von der Ordnung $O(n^2)$. Dieser Extremfall kommt in der Praxis selten vor, so daß man Quicksort als Verfahren der Ordnung $O(n*\log_2(n))$ einstuft. Er tritt auf, wenn man als »mittleres Element« einfach immer das erste von jedem Teilfeld nimmt und das Feld vorher schon sortiert oder in großen Teilen sortiert war. Dann ist nämlich das erste Element immer das kleinste.

Zwischen $n*\log_2(n)$ und n^2 liegt insbesondere bei großen n ein beachtlicher Unterschied, den man den Formeln nicht sofort ansieht. Dazu zeigen wir folgende Grafik, die auf gemessenen Laufzeiten (siehe Zahlenmaterial am Kapitelende) basiert.

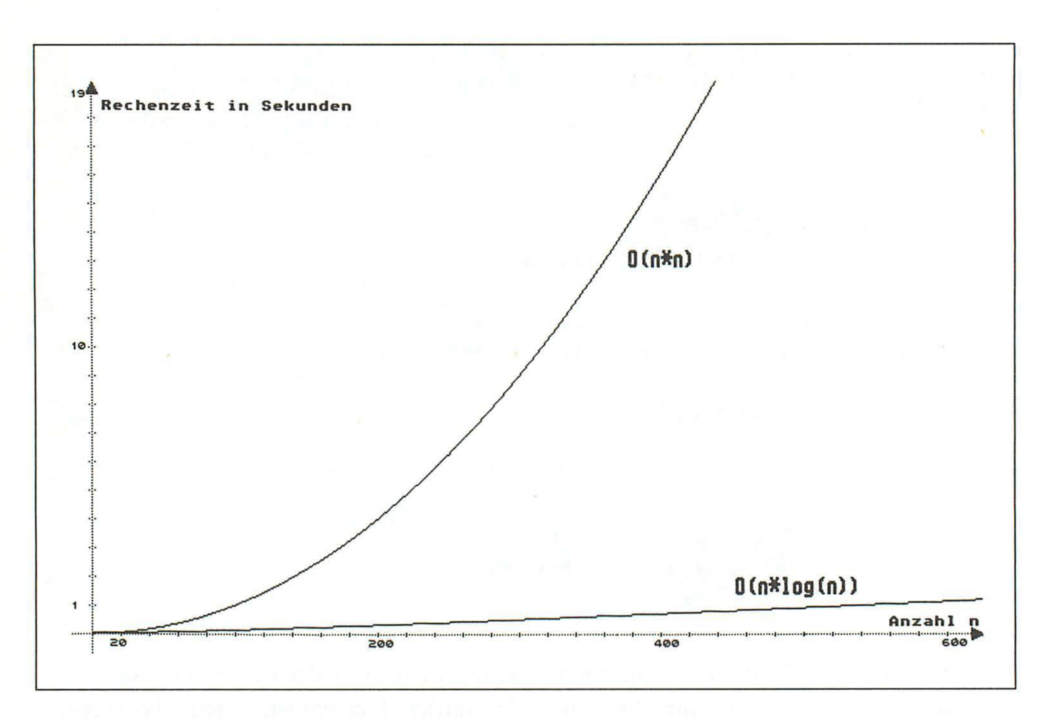


Bild 2.1: Algorithmen der Ordnung $O(n^2)$ und $O(n \log n)$

Sortieren durch Austauschen ist nur für sehr kleine n (etwa n<5) dem Quicksort-Algorithmus überlegen, da hier der Aufwand für die Zerlegung in Teilfelder und die damit verbundenen Prozeduraufrufe (bei der Rekursion) unverhältnismäßig groß sind. Wir nehmen dies zum Anlaß, Quicksort zu verbessern.

Optimierter Quicksort-Algorithmus und der externe Modul »Felder«

Folgende Tricks werden zur Beschleunigung vorgenommen:

1. Wenn die Feldlänge der zu sortierenden Teilfelder unter 5 sinkt, rufen wir nicht mehr `QuickSort`, sondern das einfache `sortiere` auf.

2. Bei einem Feldzugriff wie `feld[a]` muß jeweils eine Multiplikation durchgeführt werden, denn die Adresse von `feld[a]` wird aus der Basisadresse `feld[0] + a *TSIZE(FeldElementTyp)` errechnet. Bei den beiden WHILE-Schleifen wird das Feld jedoch nur sequentiell durchlaufen, d. h. die Adressen werden nur um `TSIZE(FeldElementTyp)` höher bzw. niedriger gesetzt. Wir nutzen dies aus, indem wir zwei Zeiger `pa` und `pb`, die auf `feld[a]` und `feld[b]` zeigen, jeweils erhöhen bzw. erniedrigen. Eine Addition bzw. Subtraktion ist schneller als eine Multiplikation.

Quicksort soll aber nicht nur geschwindigkeitsmäßig »getuned« werden. Eine weitere Zielsetzung ist die Schaffung eines allgemein verwendbaren Moduls Felder, der die Prozeduren `QuickSort` und `binaerSuche` zusammenfaßt. Dieser Modul soll unabhängig vom zu sortierenden Feldtyp sowie von den Feldgrenzen sein. Mit dieser Flexibilität kann man u.a. in einem Programm zwei verschiedene Felder mit verschiedenen Vergleichsprozeduren behandeln, z. B. Kunden (Schlüssel ist die Kundennummer vom Typ CARDINAL) und Artikel (Schlüssel ist die Artikelbezeichnung vom Typ String). Dazu sind folgende Änderungen nötig:

3. In der Parameterliste muß dann die Basisadresse des Feldes, die Anzahl der zu sortierenden Feldelemente sowie die Größe einer Feldkomponente in Bytes übergeben werden.
4. Die »kleiner«-Relation für die Schlüsselvergleiche muß vom aufrufenden Modul bereitgestellt werden, da sie sich explizit auf den jeweiligen Datentyp des Feldes bezieht.
5. Quicksort ruft sich zweimal selbst auf. Um das zeitaufwendige Kopieren der nunmehr recht langen Parameterliste bei der Rekursion zu vermeiden, wird das eigentliche Sortieren in einer lokalen Prozedur `qSort` mit einer minimalen Parameterliste abgehandelt. Außerdem sorgen wir dafür, daß der Rekursionsstack nicht überläuft, in dem zunächst immer das kleinere der beiden Teilfelder weiterbehandelt wird.
6. Da der Modul typenunabhängig arbeiten soll, kann man sich in der Vertauschprozedur nicht auf einen bestimmten Typ festlegen. Deshalb verwenden wir die universelle (und schnelle) Prozedur `SwapN` aus dem Modul `LowLevel`, der im Kapitel 3 beschrieben wird. Deshalb ist `LowLevel` vor `Felder` zu kompilieren.
7. Auch für die Zwischenspeicherung des Vergleichselementes `feld[(a+ b)DIV2]` müssen wir einen Trick anwenden, da unserem Modul dessen Datentyp nicht bekannt ist. Wir kopieren hierzu diese Variable byteweise mit der Prozedur `CopyN` aus dem Modul `LowLevel` auf den Heap. Hierzu muß zunächst der nötige Speicherplatz anfangs reserviert und nach Beendigung des Sortierens wieder freigegeben werden. Dies ist neben dem Aufruf der lokalen Prozedur `qSort` die einzige Aufgabe der übergeordneten Prozedur `QuickSort`.

```
DEFINITION MODULE Felder;

FROM SYSTEM IMPORT ADDRESS;

TYPE
kleinerProzedur = PROCEDURE(ADDRESS,ADDRESS): BOOLEAN;

PROCEDURE QuickSort(
      FeldAddresse : ADDRESS;           (* Basisadresse des zu sort. Feldes  *)
      Anzahl       : CARDINAL;          (* Zahl der zu sortierenden Elemente *)
```

```
    groesse      : LONGCARD;          (* Anz. d. Bytes einer Feldkomponente *)
    kleiner      : kleinerProzedur); (* Vergleichsproz. für die Schlüssel *)

PROCEDURE binaerSuche(
            FeldAddresse : ADDRESS;
            pGesucht     : ADDRESS;     (* Adresse des gesuchten Schlüssels *)
            links,rechts : CARDINAL;
            groesse      : LONGCARD;
            kleiner      : kleinerProzedur): CARDINAL;
        (*
         * Rückgabewert ist der entsprechende Feldindex, wenn der
         * Schlüssel gefunden wurde, andernfalls 'rechts + 1'
         *)

END Felder.
```

Die Prozeduren `CopyN` und `SwapN` importieren wir aus Geschwindigkeitsgründen aus dem Maschinensprache-Modul `LowLevel`. Natürlich kann man sie auch durch eine reine Modula-Version ersetzten:

```
PROCEDURE CopyN(von, nach : ADDRESS; groesse : LONGCARD);
VAR i: LONGCARD;
    pVon, pNach: POINTER TO BYTE;
BEGIN
  pVon := von; pNach := nach;
  FOR i := 1D TO groesse DO
    pNach^ := pVon^;
    INC(pNach); INC(pVon)
  END
END CopyN;

PROCEDURE SwapN(adr1, adr2 : ADDRESS; groesse : LONGCARD);
VAR i      : LONGCARD;
    hilf   : BYTE;
    p1, p2 : POINTER TO BYTE;

BEGIN
  p1 := adr1; p2 := adr2;
  FOR i := 1D TO groesse DO
    hilf := p2^; p2^ := p1^; p2^ := hilf;
    INC(p1); INC(p2)
  END
END SwapN;
```

Damit kann im Implementationsmodul der Import von `LowLevel` entfallen, allerdings ist der Datentyp `BYTE` aus `SYSTEM` zu importieren.

```
IMPLEMENTATION MODULE Felder;

FROM SYSTEM   IMPORT ADDRESS;
FROM Storage  IMPORT ALLOCATE, DEALLOCATE;
FROM LowLevel IMPORT CopyN, SwapN;

PROCEDURE QuickSort(
        FeldAdresse : ADDRESS;
        Anzahl      : CARDINAL;
        groesse     : LONGCARD;
        kleiner     : kleinerProzedur);

VAR pa,pb,pm: ADDRESS;

    PROCEDURE ZeigerElement(i: CARDINAL): ADDRESS;
    BEGIN
        RETURN FeldAdresse + LONG(i) * groesse
    END ZeigerElement;

(* -------------------------------------------
* zu lahm! braucht bei 1000 REAL's 2:12 min:sec!
    PROCEDURE PrimitivSort1 (li,re: CARDINAL);
    VAR i,j: CARDINAL;
        pi: ADDRESS;
    BEGIN
        FOR i := li TO re-1 DO
            pi := ZeigerElement(i);
            FOR j := i+ 1 TO re DO
                IF kleiner(ZeigerElement(j), pi) THEN
                    SwapN(ZeigerElement(j), pi, groesse) END
            END
        END
    END PrimitivSort1;
 ------------------------------------------------ *)

(* --- bei 1000 REAL's: 1:35 min:sec ---> *)
    PROCEDURE PrimitivSort(li,re: CARDINAL);
    VAR i,j:    CARDINAL;
        pmin:   ADDRESS;
    BEGIN
        FOR i := li TO re-1 DO
```

```
            pmin := ZeigerElement(i);
            FOR j := i+1 TO re DO
                IF kleiner(ZeigerElement(j), pmin) THEN
                    pmin := ZeigerElement(j) END
            END;
            SwapN(ZeigerElement(i),pmin,groesse)
        END
    END PrimitivSort;

(* ------------------------------------------------------------------- *)
    PROCEDURE qSort(li,re: CARDINAL);
    VAR a,b: CARDINAL;
    BEGIN
        IF re-li < 4 THEN        (* ab 4 Elementen lohnt sich PrimitivSort! *)
            PrimitivSort(li,re)
        ELSE
            a := li; b := re;
            CopyN(ZeigerElement((li+re) DIV 2), pm, groesse);
            pa := ZeigerElement(a);
            pb := ZeigerElement(b);
            REPEAT
                WHILE kleiner(pa,pm) DO INC(a); INC(pa,groesse) END;
                WHILE kleiner(pm,pb) DO DEC(b); DEC(pb,groesse) END;
                IF a <= b THEN
                    SwapN(pa,pb,groesse);
                    INC(a); INC(pa,groesse);
                    DEC(b); DEC(pb,groesse) END
            UNTIL a > b;
            IF b-li < re-a THEN         (* zuerst kleineres Feld sortieren *)
                IF li < b THEN qSort(li,b) END;
                IF a < re THEN qSort(a,re) END
            ELSE
                IF a < re THEN qSort(a,re) END;
                IF li < b THEN qSort(li,b) END
            END
        END
    END qSort;

BEGIN  (* QuickSort *)
    ALLOCATE(pm,groesse);
    qSort(0, Anzahl-1);
    DEALLOCATE(pm,groesse)
END QuickSort;

(* ------------------------------------------------------------ --------*)
```

```
PROCEDURE binaerSuche(
        FeldAdresse   : ADDRESS;
        pGesucht      : ADDRESS;
        links, rechts : CARDINAL;
        groesse       : LONGCARD;
        kleiner       : kleinerProzedur): CARDINAL;
VAR
    m, li, re : CARDINAL;

    PROCEDURE gleich(p1,p2: ADDRESS): BOOLEAN;
    BEGIN
        RETURN NOT (kleiner(p1,p2) OR kleiner(p2,p1))
    END gleich;

    PROCEDURE ZeigerElement(i: CARDINAL): ADDRESS;
    BEGIN
        RETURN FeldAdresse + LONG(i) * groesse
    END ZeigerElement;

BEGIN (* binaerSuche *)
  li := 0; re := rechts - links;
  WHILE li < re DO
    m := (li + re) DIV 2;
    IF kleiner (ZeigerElement(m),pGesucht)
        THEN li := m+1
        ELSE re := m
    END
  END;
  IF gleich(pGesucht,ZeigerElement(li)) THEN RETURN links+li;
  ELSE RETURN rechts+1 END
END binaerSuche;

END Felder.
```

Wir demonstrieren die Verwendung des Moduls `Felder` zum Sortieren eines Feldes von reellen Zahlen:

```
MODULE FeldDemo;

FROM SYSTEM    IMPORT ADDRESS, ADR, TSIZE;
FROM InOut     IMPORT Read, WriteCard, WriteFix, WriteLn, WriteString;
FROM RandomGen IMPORT Randomize, Random;
FROM Felder IMPORT QuickSort;
```

```
CONST max = 20;

TYPE ZeigerTyp = POINTER TO REAL;

(*--------------------------------------------------------
: Die Vergleichsfunktion für REAL-Zahlen
: Wird der QuickSort-Funktion als Parameter übergeben
: Kann für jeden Datentyp analog geschrieben werden
+ -------------------------------------------------------*)
PROCEDURE realKleiner(a,b: ADDRESS): BOOLEAN;
VAR pa, pb: ZeigerTyp;
BEGIN
  pa := a;
  pb := b;
  RETURN pa^ < pb^
END realKleiner;

(* -------------------------------------------------------
: Druckt ein Feld von REAL-Zahlen beliebiger Länge aus
+ -------------------------------------------------------*)
PROCEDURE DruckeFeld(VAR feld: ARRAY OF REAL);
VAR i: CARDINAL;
BEGIN
  FOR i := 0 TO HIGH(feld) DO
    WriteLn; WriteCard(i,3); WriteFix(feld[i],9,2)
  END;
  WriteLn;
END DruckeFeld;

VAR
    UnserFeld : ARRAY[1..max] OF REAL;
    i         : CARDINAL;
    taste     : CHAR;

BEGIN
  (* -----   Feld mit Zufallszahlen belegen   ----- *)
  Randomize(1234567);
  FOR i := 1 TO max DO UnserFeld[i] := 100.0 * Random() END;
  DruckeFeld(UnserFeld);                 (* Feld vor dem Sortieren drucken *)
  QuickSort( ADR(UnserFeld), max, TSIZE(REAL), realKleiner);
  DruckeFeld(UnserFeld);                 (* Feld ist jetzt sortiert *)
  WriteLn; WriteString("Bitte Taste drücken");
  Read(taste);
END FeldDemo.
```

Die Prozedur `realKleiner` arbeitet in unserem Beispiel auf reellen Zahlen. In entsprechender Weise läßt eine Boolesche Prozedur `kleiner` für jeden Datentyp implementieren. Wichtig dabei ist folgendes:

1. `kleiner(a,a) = FALSE`

2. Entweder gilt `kleiner(a,b)` oder `kleiner(b,a)`

3. Gilt `kleiner(a,b)` und `kleiner(b,c)`, so muß auch `kleiner(a,c)` gelten.

Als Übung für den Leser schlagen wir vor, ein Feld vom eingangs definierten Kundentyp zunächst nach der Kundennummer und anschließend nach dem Namen zu sortieren, das zuerst sortierte Feld in eine weitere Feldvariable zu speichern und so eine Möglichkeit der schnellen Suche nach beiden Schlüsseln zu realisieren.

Wir haben einmal die Laufzeiten für Quicksort und Sortieren durch Auswahl bei unserem Modul `Felder` bei verschiedenen Feldgrößen gemessen. Die Unterschiede sind sehenswert: z.B. bei einem Feld von 1000 REAL-Zahlen ist Quicksort ca. 50mal schneller! Die übrigen Ergebnisse zeigt die abschließende Tabelle:

Feldelemente n	**Quicksort**	**Sortieren durch Auswahl**
3	0.0028 s	0.0026 s
5	0.004 s	0.004 s
10	0.009 s	0.013 s
31	0.036 s	0.1 s
100	0.17 s	1.0 s
316	0.57 s	9.7 s
1000	2.0 s	94 s
3162	7.8 s	900 s(= ¼ Stunde)
10000	28.0 s	9000 s(=2 ½ Stunden!)

Man erkennt, daß die Zeit für das Sortieren mit unserem optimierten Quicksortalgorithmus etwa $0.0002*n*\log_2(n)$s und die für das Sortieren durch Austausch etwa $0.0001*n^2$ Sekunden beträgt, wie es die obige Grafik zeigt. Der Verbesserungsfaktor ist also $n/(2*\log_2(n))$ und wächst daher stark mit steigender Anzahl der Feldelemente n.

Der Aufwand hat sich gelohnt: Wir verfügen nun über einen allgemein brauchbaren, schnellen Modul, den man für die immer wiederkehrenden Arbeiten mit Feldern lediglich in die eigenen Programme zu importieren hat! Mit dieser Philosophie werden in den folgenden Kapiteln Module für dynamische Datenstrukturen und für die Dateiverwaltung kreiert.

2.2 Verzeigerte Strukturen

In Abschnitt 1.6.6 haben wir das Zeigerkonzept eingeführt und seine Vorteile benannt. Hier geht es nun zur Sache! Ziel dieses Abschnitts ist es, mit drei wichtigen verzeigerten Datenstrukturen – Stapel, Schlange und Baum – vertraut zu machen.

Wir werden hierzu wie im Abschnitt über Felder allgemein verwendbare externe Module schreiben, die für alle mögliche Datentypen konkreter Anwendungsfälle funktionieren (Datenabstraktion). Ihre Verwendung wird anschließend in Beispielprogrammen demonstriert.

2.2.1 Die Datenstruktur »Stapel«

Man stelle sich einen Stapel – zum Beispiel von Bierkästen – vor. Auf den Stapel kann man nur oben einen weiteren Kasten auflegen. Auch das Abnehmen ist nur elementweise von oben her möglich, solange bis der Stapel leer ist. Ein Zugriff auf den fünften Kasten in einem Stapel von 10 Kästen wäre verhängnisvoll, das lassen wir lieber.

Man hat also immer nur Zugriff auf das zuletzt abgelegte Element. Das Abstapeln wird in umgekehrter Reihenfolge erledigt, wie aufgelegt wurde . Man spricht daher vom LIFO-Prinzip (**L**ast **I**n – **F**irst **O**ut, »zuerst rein – zuletzt raus«). Um solch eine Stapelstruktur in Modula zu simulieren, braucht man einen Zeiger, der immer auf das zuletzt abgelegte Element zeigt. Damit weiß man, ab wo man weiter Aufstapeln oder Abnehmen kann, nennen wir diesen Zeiger `StapelPtr` (»Ptr« steht für *pointer*, = »Zeiger«).

Wenn noch kein Stapel da ist, setzen wir `StapelPtr` auf `NIL` (er zeigt also auf »Nichts«). Diese Initialisierung des Stapels erledigt die Prozedur `Einrichten`.

Die Prozedur, die bestimmte Objekte (wir abstrahieren nun leider vom anschaulichem Eingangsbeispiel) auf den Stapel legt, nennen wir nicht etwa »Auflegen«, sondern nach altem Brauch »Push«. `Push` muß mit `Storage.ALLOCATE` Speicher auf dem Heap reservieren und `StapelPtr` aktualisieren.

Entsprechend wird ein Objekt mit `Pull` vom Stapel geholt. Hier ist `StapelPtr` zu erniedrigen und der Speicherbereich mit `Storage.DEALLOCATE` wieder freizugeben. Ein Zugriff mit `Pull` auf einen leeren Stapel wäre fatal, man würde in undefinierte Speicherbereiche greifen. Daher deklarieren wir eine Boolesche Funktion `leer`, die Auskunft gibt, ob der Stapel leer ist. Diese vier Prozeduren reichen aus, um einen Stapel (engl. *stack*) zu verwalten.

Stapel spielen eine wichtige Rolle beim Kompilierungsvorgang:

- Als wichtigstes Hilfsmittel ist die Benutzung eines Stapels beim Aufruf von Prozeduren zu nennen, weil hierbei die Rücksprungadressen auf einen Stapel (den Hardware-Stack) abgelegt werden. Die Parameter einer Prozedur werden beim Aufruf der Prozedur auf einen

Stapel gelegt. Die Prozedur holt sich als erstes diese Parameter (in umgekehrter Reihenfolge) wieder von diesem Stapel ab.

- Bei der Bearbeitung von arithmetischen Ausdrücken werden in den Termen von links nach rechts die Operanden und Operatoren auf einem Stapel zwischengespeichert, falls sie in der hierarchischen Ordnung noch nicht auszuführen sind.

Unser erstes Beispiel ist etwas bescheidener. Es zeigt die Addition zweier natürlicher Zahlen mit beliebig großer Stellenzahl.

Es werden drei Stapel benötigt: je einer für die beiden Summanden und einer für die Summe. Gestapelt werden nur die Ziffern 0 bis 9. Mit der Prozedur `ZahlEingeben` wird für jeden Summanden ein Stapel erzeugt. Die Prozedur `Addieren` holt anschließend die letzten beiden Ziffern des Summanden, addiert diese, spaltet Endziffer und Übertrag ab und speichert die Endziffer im Stapel `ergebnis` ab. Nun liegen die beiden nächsten Ziffern oben auf den Summandenstapeln; sie werden abgeholt, zusammen mit dem Übertrag addiert und wie vorher weiterbehandelt. Dieser Prozeß läuft so lange ab, bis beide Stapel leer sind. Sollte vorher schon einer der beiden Summandenstapel leer sein, weil dieser Summand eine kleinere Stellenzahl hatte, so wird Null addiert. Da nun die höchstwertige Ziffer der Summe als letzte abgelegt worden ist, gibt `ZahlAusgeben` die Summe in der richtigen Reihenfolge aus (LIFO-Prinzip).

`StapelPtr` zeigt also immer auf die letzte Ziffer. Damit die anderen Ziffern auch noch zugänglich sind, muß man die Verzeigerung zum nächst tieferem Objekt ebenfalls abspeichern. Insgesamt ist also der Verbund

```
TYPE
    Knoten = RECORD
                Ziffer: [0..9];
                naechster: LIFO
             END;
```

zu stapeln. Hierbei ist

```
TYPE LIFO = POINTER TO Knoten;
```

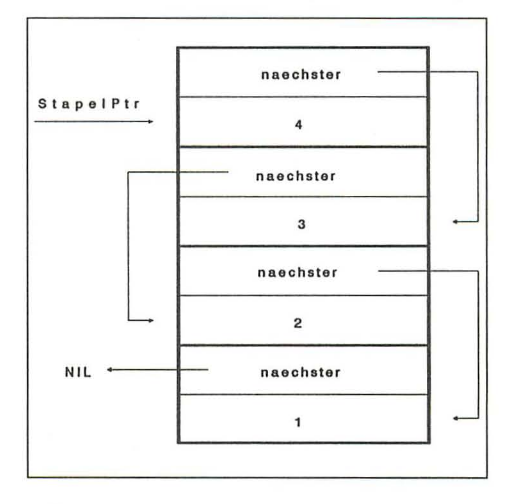


Bild 2.2: Stapel mit den Daten »1,2,3,4«

Zur Erinnerung: solche Vorwärtsreferenzen sind bei Zeigerdeklarationen ausnahmsweise erlaubt. Damit die wichtige Datenstruktur des Stapels von dem zwar anschaulichen, aber ziemlich unwichtigen Beispiel der Zahlenaddition getrennt wird, packen wir im nachfolgenden Listing die vier beschriebenen Stapelprozeduren in einen lokalen Modul »Stapel1«

```
MODULE BeliebigLangeZahlen;

FROM Storage IMPORT ALLOCATE, DEALLOCATE;        (* nur für Stapell *)
FROM InOut   IMPORT Read, Write, WriteCard, WriteString, WriteLn;

TYPE ZifferTyp = [0..9];

(* ------------------ Lokaler Modul Stapell ------------------ *)
MODULE Stapell;

IMPORT ZifferTyp, ALLOCATE, DEALLOCATE;
EXPORT LIFO, Einrichten, leer, Push, Pull;

TYPE LIFO      = POINTER TO Knoten;
     Knoten    = RECORD
                   ziffer    : ZifferTyp;
                   naechster : LIFO
                 END;

PROCEDURE Einrichten(VAR StapelPtr : LIFO);
BEGIN
  StapelPtr := NIL
END Einrichten;

PROCEDURE leer(StapelPtr: LIFO) : BOOLEAN;
BEGIN
  RETURN StapelPtr = NIL
END leer;

PROCEDURE Push(VAR StapelPtr : LIFO; z : ZifferTyp);
VAR p  : LIFO;
BEGIN
  ALLOCATE(p,SIZE(p^));
  p^.ziffer := z;
  p^.naechster := StapelPtr;
  StapelPtr := p;
END Push;

PROCEDURE Pull(VAR StapelPtr : LIFO; VAR z : ZifferTyp);
VAR p : LIFO;
BEGIN
  IF NOT leer(StapelPtr) THEN
    z := StapelPtr^.ziffer;
    p := StapelPtr^.naechster;
```

```
    DEALLOCATE(StapelPtr,SIZE(StapelPtr^));
    StapelPtr := p
  END;
END Pull;

END Stapel1;
(* --------------------------------------------------------------- *)

PROCEDURE ZahlEingeben(VAR zahl : LIFO);
VAR ch : CHAR;
BEGIN
  Einrichten(zahl);
  LOOP
    Read(ch);
    CASE ch OF
      15C      : EXIT |
      "0".."9" : Push(zahl,ORD(ch) - ORD("0"));
    ELSE Write(7C) END;
  END
END ZahlEingeben;

PROCEDURE ZahlAusgeben(zahl : LIFO);
VAR ziffer : ZifferTyp;
BEGIN
  WHILE NOT leer(zahl) DO
    Pull(zahl,ziffer);
    WriteCard(ziffer,1)
  END
END ZahlAusgeben;

PROCEDURE Addieren(zahl1, zahl2 : LIFO; VAR summe : LIFO);
VAR ziffer1, ziffer2, ziffer, uebertrag : ZifferTyp;
    hilf                                : CARDINAL;
BEGIN
  Einrichten(summe);
  uebertrag := 0;
  WHILE NOT (leer(zahl1) & leer(zahl2)) DO
    IF leer(zahl1) THEN ziffer1 := 0 ELSE Pull(zahl1,ziffer1) END;
    IF leer(zahl2) THEN ziffer2 := 0 ELSE Pull(zahl2,ziffer2) END;
    hilf := ziffer1 + ziffer2 + uebertrag;
    ziffer := hilf MOD 10;
    uebertrag := hilf DIV 10;
    Push(summe,ziffer)
  END;
```

```
  IF uebertrag > 0 THEN Push(summe,uebertrag) END
END Addieren;

VAR summand1, summand2, ergebnis : LIFO;
    antwort                      : CHAR;

BEGIN  (* BeliebigLangeZahlen *)
  WriteLn; WriteString("Demonstration eines Stapels ");
  WriteString("am Beispiel der Addition von langen Zahlen"); WriteLn;
  REPEAT
    WriteLn; WriteString("1. Zahl (>0, Länge beliebig): ");
    ZahlEingeben(summand1);
    WriteLn; WriteString("2. Zahl (>0, Länge beliebig): ");
    ZahlEingeben(summand2);
    Addieren(summand1,summand2,ergebnis);
    WriteLn; WriteString("Die Summe beträgt             : ");
    ZahlAusgeben(ergebnis);
    WriteLn; WriteLn; WriteString("Noch einmal (j/n) ? ");
    Read(antwort); antwort := CAP(antwort)
  UNTIL antwort = "N"
END BeliebigLangeZahlen.
```

Der externe Modul »Stapel«

Wie man an unserem Beispielprogramm erkennt, ist die Programmierung und Handhabung eines Stapels recht einfach.

Leider ist unser lokaler Modul `Stapel1` noch nicht allgemein brauchbar. Für die abgespeicherten Inhalte sind hier nur die Ziffern 0 bis 9 zugelassen (vgl. Deklaration von `LIFO`).

Durch folgenden Kunstgriff erreichen wir es nun, diese Typabhängigkeit zu durchbrechen: Wir stapeln gar nicht mehr die eigentlichen Daten auf, sondern nur noch die Zeiger auf ihre Adresse! Die Daten selbst werden an anderer Stelle auf den Heap gelegt.

Beim Abholen der Daten findet man auf dem Stapel also deren Adresse. Da beliebige Datentypen zugelassen sind, benötigt man aber noch eine Information über die Anzahl der Bytes einer abgespeicherten Variable. Diesen Wert muß man also ebenfalls mit auf den Stapel geben. Alle Informationen speichern wir in einem Verbund auf dem Stapel:

```
TYPE
    LIFO = POINTER TO Kopf;
    Kopf = RECORD
               ByteZahl : CARDINAL;
               Eintrag  : ADDRESS;
               naechster: LIFO
           END;
```

Da beim Abspeichern sowohl eines Stapelobjekts als auch der eigentlichen Daten der Heap in der Reihenfolge von unten nach oben wächst, ergibt sich nach zweimaligem `Push` von Daten das nebenstehende Bild

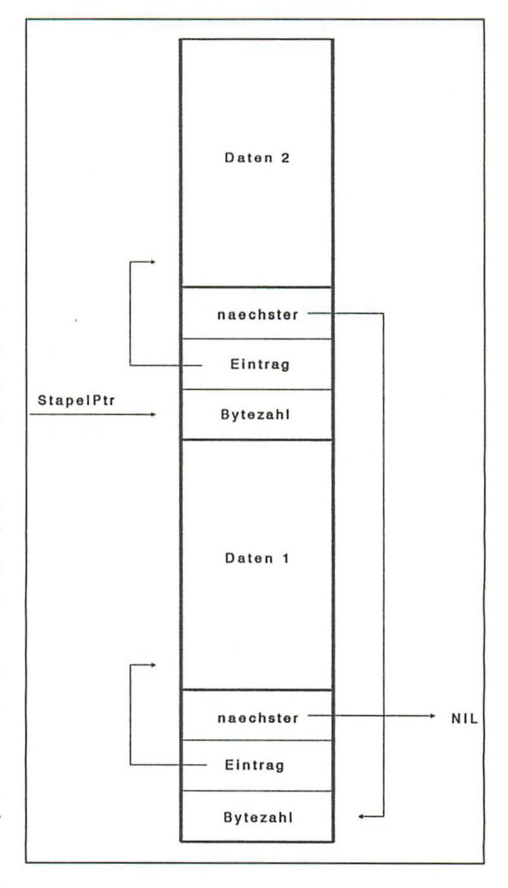


Bild 2.3: Heap mit zwei Daten

Die in der Zeichnung markierten Teile gehören zur Stapelverwaltung: sie bilden den Kopf, der für jeden Datensatz mitangelegt wird. Den Rest bilden die eigentlichen Daten, die jeweils über den *headern* (»Köpfen«) liegen. Man erkennt, daß mit dem Fortschritt der Allgemeinnützigkeit der Verwaltungsaufwand steigt (hier gibt es Parallelen zu anderen Verwaltungen).

Die neue Prozedur `Push` muß nun zweimal Speicherplatz anfordern: einmal für den Kopf und zum anderen für die eigentlichen Daten. Ebenso wird in `Pull` zweimal Speicherplatz freigegeben.

Wir formulieren nun einen externen Modul `Stapel`. Die bisher geschilderten Einzelheiten sind nur wichtig zum Verständnis des Implementationsmoduls. Der Programmierer, der aber beim Gebrauch des Moduls andere Probleme im Kopf hat, muß von diesen Einzelheiten entlastet werden. Er braucht nicht zu wissen, wie die Abspeicherung vor sich geht (»Geheimnisprinzip«). Ja, er braucht nicht einmal zu wissen, mit welchen Datenstrukturen dies im einzelnen realisiert wird. Krasser gesagt, das geht ihn überhaupt nichts an! Also verbergen wir nicht nur die Implementierung der Prozedur im Implementationsmodul, sondern auch noch den zugrunde liegenden Datentyp `LIFO` durch einen »opaken« Export.

Wir geben dem Programmierer über das Definitionsmodul also einen Datentyp:

```
TYPE LIFO;
```

und einen Satz Prozeduren zur Verfügung, die es ihm gestattet, damit umzugehen. Mehr braucht er nicht. Er sieht so genau, was er benutzen kann, und es besteht keine Gefahr, daß er durch unvorsichtiges hantieren mit diesem Datentyp die komplizierte Datenstruktur »Stapel« zerstört, weil er auf die empfindlichen Elemente (wie die Verwaltungszeiger) nicht zugreifen kann. Außerdem bleibt der Definitionsmodul so schön übersichtlich. Im Implementationsmodul muß sich natürlich die vollständige Deklaration befinden.

Einen solchen Modul nennt man einen »abstrakten Datentyp« (ADT). Wir finden unsere eingangs beschriebenen Prozeduren wieder:

```
DEFINITION MODULE Stapel;    (* ADT = Abstrakter Datentyp *)

FROM SYSTEM IMPORT BYTE;

TYPE LIFO;     (* Opaker Export *)

PROCEDURE Einrichten(VAR StapelPtr : LIFO);
        (*
         * Richtet einen leeren Stapel ein, auf den "StapelPtr" zeigt.
         *)

PROCEDURE leer(StapelPtr: LIFO) : BOOLEAN;
        (*
         * Ergibt TRUE, wenn der Stapel leer ist.
         *)

PROCEDURE Push(VAR StapelPtr : LIFO; inhalt : ARRAY OF BYTE);
        (*
         * Legt 'inhalt' auf den Stapel.
         *)

PROCEDURE Pull(VAR StapelPtr : LIFO; VAR inhalt : ARRAY OF BYTE);
        (*
         * Holt 'inhalt' vom Stapel.
         * Der Stapel darf nicht leer sein (voher mit 'leer' testen).
         * Die Variable 'inhalt' muß dieselbe Bytezahl wie bei 'Push'
         * haben.
         *)

END Stapel.
```

Zur Implementation ist bereits alles Nötige gesagt worden. Interessant dürfte auch noch die Prozedur `CopyN` sein, die Variablen beliebigen Typs und beliebiger Größe kopiert. Im Gegensatz zum vorigen Abschnitt, wo hierfür auf den Assemblersprachen-Modul `LowLevel` zurückgegriffen wurde, stellen wir hier eine Version vor, die voll in Modula geschrieben ist.

```
IMPLEMENTATION MODULE Stapel;

FROM SYSTEM  IMPORT BYTE, ADR, ADDRESS;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;

TYPE LIFO     = POINTER TO kopf;
     kopf     = RECORD
                  ByteZahl  : CARDINAL;
                  Eintrag   : ADDRESS;
                  naechster : LIFO
                END;

PROCEDURE Einrichten(VAR StapelPtr : LIFO);
BEGIN
  StapelPtr := NIL
END Einrichten;

PROCEDURE leer(StapelPtr: LIFO) : BOOLEAN;
BEGIN
  RETURN (StapelPtr = NIL)
END leer;

PROCEDURE CopyN(von, nach : ADDRESS; groesse : CARDINAL);
VAR i : CARDINAL;
    pVon, pNach : POINTER TO BYTE;
BEGIN
  pVon := von; pNach := nach;
  FOR i := 1 TO groesse DO
    pNach^ := pVon^;
    INC(pNach); INC(pVon)
  END
END CopyN;

PROCEDURE Push(VAR StapelPtr : LIFO; inhalt : ARRAY OF BYTE);
VAR
    i,anz   : CARDINAL;
    p       : LIFO;
BEGIN
  anz:=HIGH(inhalt);
```

```
  ALLOCATE(p,SIZE(p^));
  WITH p^ DO
    ByteZahl:=anz;
    ALLOCATE(Eintrag,LONG(anz + 1));
    CopyN(ADR(inhalt),Eintrag,anz + 1);
    naechster:=StapelPtr
  END;
  StapelPtr:=p
END Push;

PROCEDURE Pull(VAR StapelPtr : LIFO; VAR inhalt : ARRAY OF BYTE);
VAR i,anz : CARDINAL;
    p     : LIFO;
BEGIN
  anz:=HIGH(inhalt);
  IF leer(StapelPtr) OR (StapelPtr^.ByteZahl # anz) THEN HALT END;
  p:=StapelPtr;
  WITH p^ DO
    StapelPtr:=naechster;
    CopyN(Eintrag, ADR(inhalt),anz + 1);
    DEALLOCATE(Eintrag,LONG(anz+ 1))
  END;
  DEALLOCATE(p,SIZE(p^))
END Pull;

END Stapel.
```

Anwendung des externen Moduls »Stapel«

In einem kleinem Beispiel soll nun demonstriert werden, wie man mit dem externen Modul `Stapel` umgeht.

Wie bereits erwähnt, werden Stapel beim Abarbeiten rekursiver Prozeduren verwendet. Der Programmierer kann nun auch selber rekursive Prozeduren iterativ formulieren, indem er zum Ablegen der Parameter einen Stapel benutzt. Wenigstens auf diese Art läßt sich jede rekursive Prozedur in eine iterative umschreiben. Dieses Verfahren wird in dem folgenden Programm am Beispiel der schon bekannten Prozedur `QuickSort` demonstriert.

Die Prozedur `push2` speichert die beiden Feldgrenzen auf einen Stapel, `pull2` holt sie dort wieder ab. Die Prozedur `Quicksort` selbst wird nur geringfügig verändert (vergleichen Sie mit der ersten Version). Zur Demonstration des Algorithmus wird ein Feld von 100 Zufallszahlen sortiert.

```
MODULE StapelDemoMitQuickSortIterativ;

IMPORT Stapel;
FROM InOut     IMPORT Read, WriteCard, Write-String, WriteLn;
FROM RandomGen IMPORT RandomCard, Randomize;

CONST max = 100;

TYPE DoppelWort = RECORD i1,i2 : CARDINAL END;
VAR  feld       : ARRAY [1..max] OF CARDINAL;
     oben       : Stapel.LIFO;

PROCEDURE push2(a,b : CARDINAL);
VAR doppel : DoppelWort;
BEGIN
  doppel.i1 := a; doppel.i2 := b; Stapel.Push(oben,doppel)
END push2;

PROCEDURE pull2(VAR a,b : CARDINAL);
VAR doppel : DoppelWort;
BEGIN
  Stapel.Pull(oben,doppel); a := doppel.i1; b := doppel.i2
END pull2;

PROCEDURE QuickSortIterativ(links,rechts : CARDINAL);
VAR
    a,b              : CARDINAL;
    mittlerer,hilf : CARDINAL;
BEGIN
  Stapel.Einrichten(oben);
  push2(links,rechts);
  REPEAT
    pull2(links,rechts);
    a := links; b := rechts;
    mittlerer := feld[(a + b) DIV 2];
    REPEAT
      WHILE feld[a] < mittlerer DO INC(a) END;
      WHILE feld[b] > mittlerer DO DEC(b) END;
      IF a <= b THEN
        hilf := feld[a]; feld[a] := feld[b]; feld[b] := hilf;
        INC(a); DEC(b)
      END;
    UNTIL a > b;
```

```
    IF links < b THEN push2(links,b) END;
    IF a < rechts THEN push2(a,rechts) END;
  UNTIL Stapel.leer(oben)
END QuickSortIterativ;

VAR taste : CHAR;
    i     : CARDINAL;

BEGIN
  WriteString("Unsortiertes Feld:"); WriteLn;
  Randomize(0L);
  FOR i := 1 TO max DO feld[i] := RandomCard(0,999); WriteCard(feld[i],5) END;
  QuickSortIterativ(1,max);
  WriteLn; WriteString("Sortiertes Feld:"); WriteLn;
  FOR i := 1 TO max DO WriteCard(feld[i],5) END;
  Read(taste)
END StapelDemoMitQuickSortIterativ.
```

Vielleicht haben Sie Lust, unsere Stoppuhr aus Abschnitt 3.3.1 einzubauen und die Zeiten mit denen aus 2.1 zu vergleichen. Man wird erkennen, daß die rekursive Version trotzdem leicht schneller ist. Woran das liegt? Nun, bei unserer »gewaltsamen« Iteration von Quicksort geht an einigen Stellen Zeit verloren: Wo in der rekursiven Version der Selbstaufruf steht, sind in der iterativen Prozedur die Funktionen `push2` und `pull2`. Diese rufen wiederum `Stapel.Push` und `Stapel.Pull` auf. Und auch dort tut sich einiges; unter anderem rufen sie jeweils gleich zweimal `Storage.ALLOCATE` und `Storage.DEALLOCATE` auf, und diese brauchen schließlich auch ihre Zeit.

Es dürfte klar geworden sein, daß mit dem Modul `Stapel` jeder Datentyp zu behandeln ist. Zum Beispiel können in einem Programm zwei unterschiedliche Stapel mit Einträgen verschiedenen Typs bearbeitet werden. Man hat dann zwei Stapelzeiger, die auf die Spitze der jeweiligen Stapel zeigen.

Es ist aber auch möglich, in einem Stapel unterschiedliche Datentypen zu speichern, weil wir uns die Größe eines jeden Eintrags getrennt merken. Im folgenden Beispiel werden drei `REAL`-Zahlen und drei `CARDINAL`-Zahlen auf den selben Stapel gelegt.

```
MODULE StapelTest;

IMPORT Stapel;
FROM InOut IMPORT WriteReal, ReadReal, WriteCard, ReadCard, WriteLn,
                  WriteString, KeyPressed;
```

```
VAR top     : Stapel.LIFO;
    x,y,z   : REAL;
    i,j,k,l : CARDINAL;

BEGIN
  Stapel.Einrichten(top);
  WriteString("Bitte 3 relle Zahlen     : "); WriteLn;
  ReadReal(x); ReadReal(y); ReadReal(z);
  Stapel.Push(top,x); Stapel.Push(top,y); Stapel.Push(top,z);
  WriteString("Und nun 3 Kardinalzahlen : "); WriteLn;
  ReadCard(i); ReadCard(j); ReadCard(k);
  Stapel.Push(top,i); Stapel.Push(top,j); Stapel.Push(top,k);
  WriteString("Der Stapelinhalt lautet :"); WriteLn;
  FOR l := 1 TO 3 DO
    Stapel.Pull(top,i); WriteCard(i,10); WriteLn
  END;
  WHILE NOT Stapel.leer(top) DO
    Stapel.Pull(top,x); WriteReal(x,10,5); WriteLn
  END;
  REPEAT UNTIL KeyPressed()
END StapelTest.
```

Nicht erlaubt ist aber das Ablegen einer `REAL`-Zahl, die man dann mit einer `CARDINAL`-Variable abholen will (wie soll die arme Funktion `pull` auch eine `REAL`-Zahl in einem 2-Byte-`CARDINAL` unterbringen?). Einen solchen Programmierfehler bemerkt die Funktion `pull` und reagiert darauf drastisch mit einem Programmabbruch, indem sie `HALT` aufruft. Je nach System landet man dann im »Debugger« und kann mit seiner Hilfe feststellen, wo man was falsch gemacht hat. In einem funktionierenden Programm darf es zu so etwas natürlich nicht kommen. Die `HALT`-Aufrufe in unseren Implementationsmodulen dienen also nur zur Hilfe des Programmierers! Sie melden ihm unerlaubte Zugriffe.

2.2.2 Die Datenstruktur »Schlange«

Im letzten Abschnitt ging das Abholen der Daten nach dem biblischem Motto »die letzten werden die ersten sein«. Bei Warteschlangen hingegen ist die Losung: »wer zuerst kommt, mahlt zuerst«. Dieses Prinzip heißt FIFO-Prinzip (**F**irst **I**n – **F**irst **O**ut, »zuerst rein, zuerst raus«). Der geneigte Leser ahnt schon, worauf es hinaus läuft: Es soll ein abstrakter Datentyp »Schlange« geschaffen werden, der im großen und ganzen `Stapel` ähnelt, nur daß neue Daten sich »hinten« an die Schlange anstellen und die Daten von »vorne« abgeholt werden. Dies stellt sich so dar:

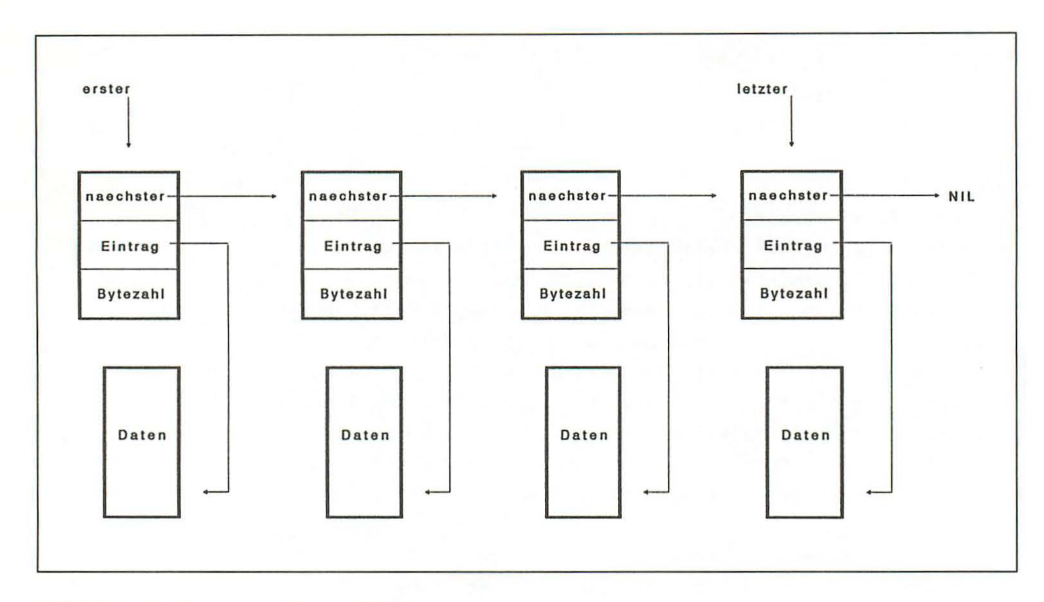


Bild 2.4: Die Datenstruktur »Schlange«

Schlangen werden zur Simulation von Warteschlangen benutzt, bei der Bearbeitung mehrerer »Jobs« in EDV-Anlagen, bei Tastaturpuffern, bei der Übertragung von Daten aus dem Kernspeicher an einen Drucker oder ein Modem.

Zu diesen Zwecken werden aber auch Strukturen von begrenzter Länge wie Arrays und der Ringpuffer (der eine »endliche« Lösung einer Schlange darstellt) eingesetzt. So kann der Tastaturpuffer voll werden; manche Rechner erzeugen dann bei weiteren Tastendrücken ein nervendes Gepiepse.

Zur Implementation einer Schlange benötigt man wiederum vier Prozeduren. `Einrichten` erzeugt eine leere Schlange mit den Zeigern `erster` und `letzter` (Anfang und Ende der Schlange).

Die Boolesche Funktion `leer` prüft, ob noch Elemente in der Schlange abgespeichert sind.

`Anfuegen` fügt ein neues Element hinten an die Schlange an; `Abholen` arbeitet genau umgekehrt: Falls noch ein Element vorhanden ist, wird es geliefert.

Man erkennt an dieser Erläuterung und der obigen Abbildung, daß die Implementation einer Schlange insofern etwas schwieriger als die des Stapels ist, als jeweils zwei Zeiger (`erster` und `letzter`) zu manipulieren sind. Da wir beim Stapel einige Erfahrung gesammelt haben, können wir ohne große Vorrede auf einen allgemein verwendbaren Modul zusteuern. Trotzdem

soll die Anwendung des Moduls `Schlange` genauso einfach sein wie `Stapel`; der Programmierer, der das Modul nur benutzen will, darf mit solchen Einzelheiten nicht belastet werden. Also kommt wieder nur der opake Export in Frage.

Wir fassen also die Zeiger `erster` und `letzter` in einem Verbund `FIFOHeader` zusammen. Dummerweise darf ein opaker Typ nur ein einziger Zeiger sein. Dann exportieren wir halt als opaken Typ `FIFO` einen Zeiger auf diesen Verbund. Der opake Typ `FIFO`, der nach außen schlicht

```
TYPE FIFO; (* sonst nix *)
```

heißt, sieht also nun so aus:

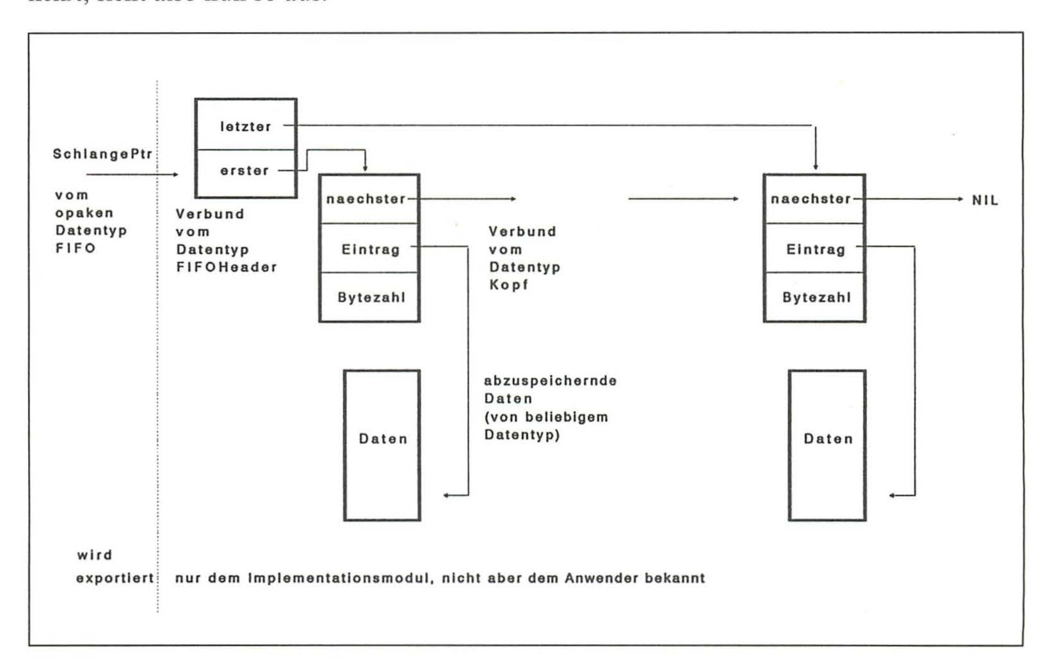


Bild 2.5: Abstrakter Datentyp »FIFO«

Die Bequemlichkeit, später nur mit dem Typ FIFO umgehen zu müssen, erkauft man sich also durch eine zusätzliche »Indirektion« (= Umweg über einen Zeiger).

```
DEFINITION MODULE Schlange;

FROM SYSTEM IMPORT BYTE;

TYPE FIFO;

PROCEDURE Einrichten(VAR SchlangePtr : FIFO);
        (*
         *  Es wird eine leere Schlange eingerichtet,
         *  auf die SchlangePtr zeigt.
         *)

PROCEDURE leer(SchlangePtr: FIFO) : BOOLEAN;
        (*
         *  Prüft, ob die Schlange leer ist.
         *)

PROCEDURE Anfuegen(SchlangePtr : FIFO; inhalt : ARRAY OF BYTE);
        (*
         *  Fügt  'inhalt' an das Ende der Schlange.
         *  'inhalt' kann von beliebigem Typ sein.
         *)

PROCEDURE Abholen(SchlangePtr : FIFO; VAR inhalt :  ARRAY OF BYTE);
        (*
         *  Holt 'inhalt' vom Kopf der Schlange.
         *  Die Schlange darf nicht leer sein (vorher mit 'leer'
         *  testen). Die Variale 'inhalt' muß dieselbe Bytezahl wie
         *  beim Abspeichern mit 'Anfuegen' haben.
         *)

END Schlange.
```

Die Implementation verläuft analog zu der des Stapels. Wir haben noch eine weitere Prozedur `checkFifo` mit aufgeführt, die wir beim Schreiben des Moduls zur Konsistenzkontrolle benutzt haben. Durch eine solche Prozedur wird eine eventuelle fehlerhafte Verzeigerung während der Entwicklungsphase erkannt. Sind die Zeiger nicht so, wie sie sein sollten, bricht das Programm mit `HALT` an einer entsprechenden Stelle ab. Ohne solche Krücken passiert es beim Umgang mit Zeigern oft, daß sich Endlosschleifen oder Zeiger ins »Nirwana« (vorzugsweise auf längst freigegebene Speicherbereiche) ergeben, die den Programmierer verzweifelt zur Reset-Taste greifen lassen.

Aus dem fertigen, getesteten Modul können solche Aufrufe natürlich entfallen; wir haben sie zur Demonstration stehen gelassen.

```
IMPLEMENTATION MODULE Schlange;

FROM SYSTEM  IMPORT BYTE, ADR, ADDRESS;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;

TYPE KopfZeiger = POINTER TO Kopf;
     FIFO       = POINTER TO FIFOHeader;
     FIFOHeader = RECORD
                    erster, letzter : KopfZeiger;
                  END;
     Kopf       = RECORD
                    ByteZahl  : CARDINAL;
                    Eintrag   : ADDRESS;
                    naechster : KopfZeiger
                  END;
(*-----------------------------------------------------------*)
(* --Interne Überprüfung auf Konsistenz, nur zu Testzwecken--*)
(* -------------- kann gelöscht werden --------------------*)
PROCEDURE CheckFifo(f:FIFO);
BEGIN
    IF f = NIL THEN HALT END;
    WITH f^ DO
        IF erster = NIL THEN
            IF letzter <> NIL THEN HALT END
        ELSE
            IF letzter = NIL THEN HALT END;
            IF letzter^.naechster <> NIL THEN HALT END
        END;
    END
END CheckFifo;

(*-----------------------------------------------------------*)
PROCEDURE Einrichten(VAR SchlangePtr : FIFO);
BEGIN
  ALLOCATE(SchlangePtr,SIZE(SchlangePtr^));
  WITH SchlangePtr^ DO
    erster :=NIL; letzter:=NIL
  END
END Einrichten;
```

```
PROCEDURE leer(SchlangePtr: FIFO) : BOOLEAN;
BEGIN
  CheckFifo(SchlangePtr);
  RETURN SchlangePtr^.erster=NIL
END leer;

PROCEDURE CopyN(von, nach : ADDRESS; groesse : CARDINAL);
VAR i : CARDINAL;
    pVon, pNach : POINTER TO BYTE;
BEGIN
  pVon := von; pNach := nach;
  FOR i := 1 TO groesse DO
    pNach^ := pVon^;
    INC(pNach); INC(pVon)
  END
END CopyN;

PROCEDURE Anfuegen(SchlangePtr: FIFO; inhalt : ARRAY OF BYTE);
VAR
    p   : KopfZeiger;
BEGIN
  CheckFifo(SchlangePtr);
  ALLOCATE(p,SIZE(p^));
  WITH p^ DO
    ByteZahl := HIGH(inhalt)+ 1;
    ALLOCATE(Eintrag,LONG(HIGH(inhalt)+ 1));
    CopyN(ADR(inhalt),Eintrag,HIGH(inhalt) + 1);
    naechster:=NIL;
  END;
  WITH SchlangePtr^ DO
    IF leer(SchlangePtr) THEN erster := p; letzter:=p
    ELSE
      letzter^.naechster := p; letzter  := p
    END
  END;
  CheckFifo(SchlangePtr)
END Anfuegen;

PROCEDURE Abholen(SchlangePtr : FIFO; VAR inhalt : ARRAY OF BYTE);
VAR
    p : KopfZeiger;
BEGIN
  CheckFifo(SchlangePtr);
  IF leer(SchlangePtr) THEN HALT END;  (* falscher Zugriff *)
```

```
    WITH SchlangePtr^ DO
      IF erster^.ByteZahl # HIGH(inhalt)+1 THEN HALT END; (* falscher Datentyp *)
      p:=erster;
      WITH p^ DO
        erster:=naechster;
        CopyN(Eintrag,ADR(inhalt), HIGH(inhalt) + 1);
        DEALLOCATE(Eintrag,ByteZahl)
      END;
      DEALLOCATE(p,SIZE(p^));
      IF erster = NIL THEN letzter:=NIL END
    END;
    CheckFifo(SchlangePtr)
  END Abholen;

END Schlange.
```

Die Benutzung des Moduls `Schlange` verläuft analog zu der von Stapel, weshalb wir hier keine Anwendungsbeispiele bis auf ein kleines Demonstrationsprogramm zeigen.

```
MODULE SchlangeTest;

FROM InOut IMPORT Read, Write, WriteLn, WriteString, ReadString;
FROM Schlange IMPORT FIFO, Einrichten, leer, Abholen, Anfuegen;

VAR schlange : FIFO;
    s        : ARRAY[0..9] OF CHAR;
    i        : CARDINAL;
    c        : CHAR;

BEGIN
  Einrichten(schlange);
  IF leer(schlange) THEN WriteString("Die Schlange ist noch leer!") END;
  WriteLn; WriteString("Geben Sie ein paar Namen ein (<RET> = Ende):");
  WriteLn;
  LOOP
    WriteString("Name: "); ReadString(s);
    IF s[0] = 0C THEN EXIT END;
    Anfuegen(schlange,s);
  END;
```

```
  WHILE NOT leer(schlange) DO
    Abholen(schlange, s);
    WriteLn; WriteString(s);
  END;
  WriteLn; WriteString("<Taste>"); Read(c)
END SchlangeTest.
```

Weitere Beispiele für den Aufruf des Moduls `Schlange` geben die Programme `Dateiverwaltung` im Abschnitt 2.3.2 und das Programm `Druck` in Abschnitt 4.4. Hier werden Namen von zu druckenden Dateien mittels einer »File-selector-box« eingegeben in eine Warteschlange gespeichert und anschließend der Reihe nach ausgedruckt. Dies entspricht einer typischen »Schlange«-Anwendung.

2.2.3 Die Datenstruktur »Baum«

Stapel und Schlangen sind zwar wichtige Elemente in der Datenverarbeitung, aber hier entscheidet sich die Ablage (und der spätere Zugriff) durch die Reihenfolge des Eintreffen der Daten.

Oft möchte man aber Daten nach einem bestimmten Kriterium »linear« ausdrucken, also etwa so sortiert, daß »oben« die kleineren Daten stehen und »unten« die größeren (Beispiel: Telefonbuch). Wir greifen also hier wieder einen Problemkreis auf, den wir schon bei Felder in Abschnitt 2.1 kennengelernt haben.

Die Daten haben neben der eigentlichen Information einen Schlüssel, nach dem sie mittels einer »kleiner-Funktions-Prozedur« sortiert werden können. Dann läßt sich schnell mit binärem Suchen ein gewünschter Datensatz aus einer großen Datenmenge herausfinden. Nun haben Felder aber den Nachteil des statischen: Ein neu ankommender Datensatz, der an der richtigen Stelle in einem sortierten Feld abgelegt werden soll, zwingt zum Umkopieren aller Feld-
elemente, die »hinter« diesem Datum liegen:

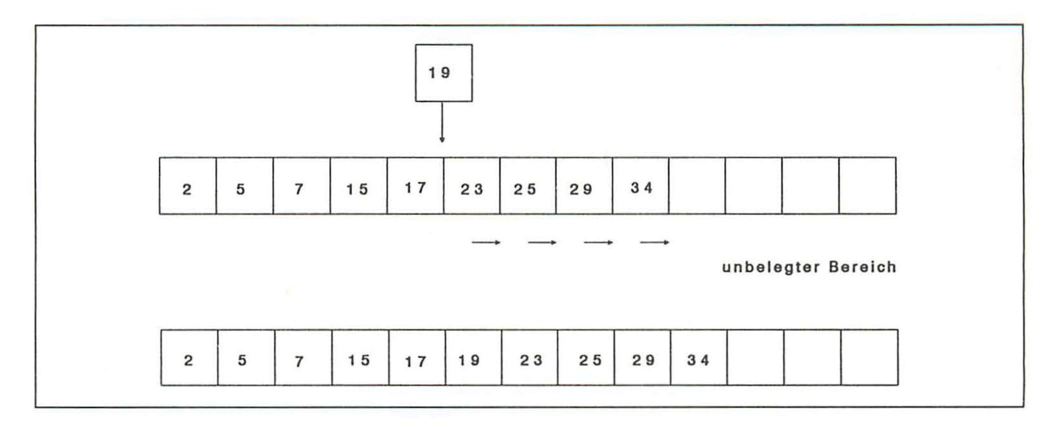


Bild 2.6: Einfügen in sortiertes Feld

In unserem Beispiel müssen die Daten 23, 25, 29,34 »umgeschaufelt« werden. In einem konkreten Anwendungsfall könnten dies mehrere Kbyte sein.

Das gleiche passiert beim Löschen eines Datensatzes. Hierzu kommt noch der bereits im Abschnitt 1.6.6 erwähnte Nachteil der festen Speicherreservierung für ein Feld: war man bei der Deklaration eines Feldes zu sparsam, so kann es beim Auftreten zusätzlicher Daten dazu kommen, daß das Feld nicht mehr zur Aufnahme ausreicht. Der rechte Feldindex muß höher gesetzt werden; das Programm ist erneut zu kompilieren. Letzteres ist nicht mehr möglich, wenn es bereits in Benutzerhand ist.

Also wählt man die Feldgrenzen gleich übermäßig groß! Das führt aber wieder zur Verschwendung von Speicherplatz, den man vielleicht an anderer Stelle gut gebrauchen könnte.

Man benötigt also eine Struktur, die die Vorteile eines Feldes mit der gewünschten Dynamik vereinigt. Diese Struktur heißt »geordneter binärer Baum mit Rückzeiger«. Ihre Kennzeichen:

- Die Daten liegen geordnet (gemäß einer »kleiner«-Funktion) vor.
- Man kann einen beliebigen Datensatz effizient einfügen oder löschen (unter Beibehaltung der Ordnung).
- Ein Datensatz mit einem bestimmten Schlüssel läßt sich schnell suchen.
- Von einem Datensatz ist beliebiges »Weiterblättern« zum Datensatz mit dem nächst größeren Schlüssel bzw. Zurückgehen auf den Datensatz mit dem nächst niedrigeren Schlüssel möglich.
- Die Ausnutzung des Speichers ist gut.

Da die Implementation mit der vollen Leistungsfähigkeit relativ schwer verständlich ist, wollen wir den Leser mit einem kleinem Programm `EinfacherBaum` in Bäume einführen.

Es handelt sich um einen Baum, dessen Inhalte nur aus Zeichen (Typ CHAR) bestehen. Das folgende Demoprogramm bietet die Möglichkeiten:

- eine geordnete Folge von Zeichen zum Beispiel
  A, C, E, G, J, R, U
  in eine ausgeglichene Baumstruktur zu überführen. Dies ist die Demonstration der Prozedur `LinearZuBaum`.
- Man kann sich den Baum auf dem Bildschirm ausdrucken lassen. Hierbei ist allerdings die Wurzel links, das heißt der Baum ist um 90° nach links gedreht.
- Man kann die Bauminhalte der Reihenfolge nach ausgeben lassen, dies ist eine Demonstration der Prozedur `BaumZuLinear`.
- Einfügen von weiteren Zeichen in den Baum ist möglich, testen Sie das aus und rufen Sie anschließend `DruckeBaum` auf!
- Die Suche nach einem abgespeicherten Zeichen ist möglich.

Wir haben also eine komplette Baumstruktur implementiert. Bis auf die Tatsache, daß nur Zeichen abgespeichert werden können und eine Prozedur zum Löschen fehlt.

```
MODULE EinfacherBaum;

FROM InOut IMPORT Read, Write, WriteString, WriteLn;
FROM Storage IMPORT ALLOCATE;

TYPE KnotenPtr = POINTER TO Knoten;
     Knoten    = RECORD
                   inhalt        : CHAR;
                   links, rechts : KnotenPtr
                 END;

VAR Wurzel : KnotenPtr;

PROCEDURE Einrichten;
BEGIN
  Wurzel := NIL
END Einrichten;

PROCEDURE Einfuegen(VAR Aktuell : KnotenPtr; inhalt : CHAR);
BEGIN
```

```
  IF Aktuell = NIL THEN                    (* Ende gefunden, Einfügen als Blatt *)
    ALLOCATE(Aktuell,SIZE(Aktuell^));
    Aktuell^.links  := NIL;
    Aktuell^.rechts := NIL;
    Aktuell^.inhalt := inhalt
  ELSIF inhalt < Aktuell^.inhalt THEN Einfuegen(Aktuell^.links,inhalt)
  ELSE Einfuegen(Aktuell^.rechts,inhalt) END
END Einfuegen;

PROCEDURE Suchen(VAR anfang : KnotenPtr; ziel : CHAR);
BEGIN
  LOOP
    IF anfang = NIL THEN EXIT END;
    IF anfang^.inhalt = ziel THEN EXIT END;
    IF anfang^.inhalt < ziel THEN anfang := anfang^.rechts
      ELSE anfang := anfang^.links END
  END
END Suchen;

PROCEDURE DruckeBaum(Aktuell : KnotenPtr; stelle : CARDINAL);
VAR i : CARDINAL;
BEGIN
  IF Aktuell # NIL THEN
    WITH Aktuell^ DO
      DruckeBaum(rechts, stelle + 1);
      FOR i := 1 TO stelle DO WriteString("     ") END;
      WriteString("|--->"); Write(inhalt); WriteLn;
      DruckeBaum(links, stelle + 1);
    END
  END
END DruckeBaum;

PROCEDURE BaumZuLinear(Anfang : KnotenPtr);
BEGIN
  IF Anfang # NIL THEN
    BaumZuLinear(Anfang^.links);
    Write(Anfang^.inhalt); Write(" ");
    BaumZuLinear(Anfang^.rechts)
  END
END BaumZuLinear;

PROCEDURE LinearZuBaum(ElementZahl : CARDINAL; VAR Anfang : KnotenPtr);
BEGIN
  IF ElementZahl = 0 THEN Anfang := NIL ELSE
```

```
      ALLOCATE(Anfang,SIZE(Anfang^));
      LinearZuBaum(ElementZahl DIV 2, Anfang^.links);
      Read(Anfang^.inhalt); Write(" ");
      LinearZuBaum(ElementZahl - 1 - ElementZahl DIV 2, Anfang^.rechts);
    END
  END LinearZuBaum;

  VAR wahl, ch : CHAR;
      p        : KnotenPtr;

  BEGIN
    WriteString("Demonstration eines einfachen Baums"); WriteLn;
    WriteString("Geben Sie 7 Buchstaben in alphabetischer Reihenfolge ein! ");
    LinearZuBaum(7,Wurzel);
    LOOP
      WriteLn;
      WriteString("E(infügen, S(uchen, B(aumdruck, L(inear, N(eu, Q(uit ");
      Read(wahl); WriteLn;
      CASE CAP(wahl) OF
        "E" : WriteString("Zeichen: "); Read(ch); Einfuegen(Wurzel,ch) |
        "S" : WriteString("Zeichen: "); Read(ch);
              p := Wurzel; Suchen(p,ch);
              IF p = NIL THEN WriteString(" Nicht im Baum!")
                         ELSE WriteString(" gefunden: "); Write(ch) END;|
        "B" : DruckeBaum(Wurzel,0) |
        "L" : BaumZuLinear(Wurzel) |
        "N" : Einrichten() |
        "Q" : EXIT
        ELSE Write(7C)
      END
    END
  END EinfacherBaum.
```

Unser kleines Programm hat nur didaktischen Wert. Der Umgang mit ihm soll eine Hilfe zur Einarbeitung sein. Wir gehen nun unser eingangs erklärtes Ziel an: die Implementation eines Baumes mit Rückzeiger, der beliebige Datentypen verwalten kann.

Einen Baum mit Rückzeiger stellt man sich so vor (die Wurzel des Baumes ist hier oben, der Baum steht also auf dem Kopf):

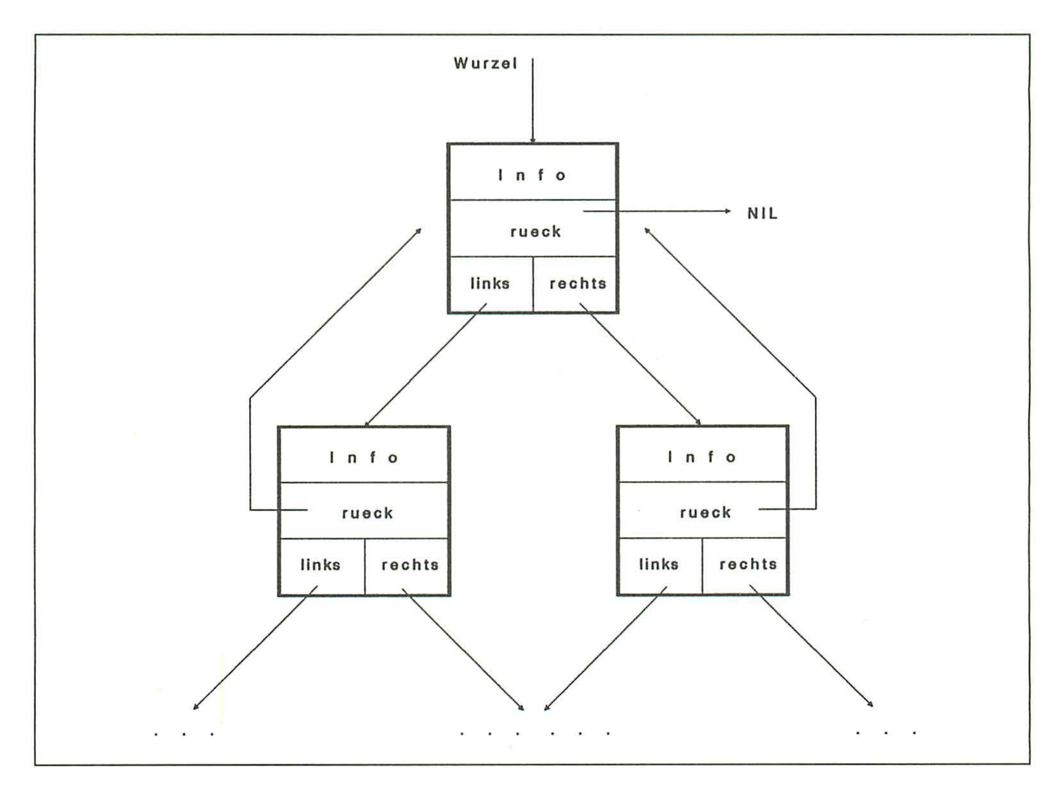


Bild 2.7: Binärer Baum mit Rückzeiger

Wie man sieht, sind nicht unbedingt alle Äste gleich lang. Als `Info` speichern wir wieder einen `Eintrag` ab, der auf die eigentlichen Daten zeigt, so wie deren Anzahl Bytes.

Wir exportieren wiederum einen opaken Typ, hier `TREE` genannt. Der aufrufende Modul muß eine Boolesche »kleiner«-Prozedur übergeben, die es dem Baum ermöglicht, festzustellen, welcher von zwei Datensätzen der kleinere ist.

Der Definitionsmodul sieht dann so aus:

```
DEFINITION MODULE Baum;

FROM SYSTEM IMPORT ADDRESS, BYTE;

TYPE
    TREE;
    KleinerProzedur = PROCEDURE( (*p1*) ADDRESS, (*p2*) ADDRESS) : BOOLEAN;
        (*
         *  Für die Festlegung der '<' - Relation zwischen zwei Schlüsseln.
```

```
     * 'p1' und 'p2' sind Zeiger auf die zu vergleichenden Daten.
     * Ist 'kleiner' vom Typ 'KleinerProzedur', so muß gelten:
     * 1. kleiner(a^,a^) = FALSE                        (nicht reflexiv)
     * 2. entweder kleiner(a^,b^) oder kleiner(b^,a^)    (asymmetrisch)
     * 3. kleiner(a^,b^) & kleiner(b^,c^) ==> kleiner(a^,c^) (transitiv)
     *)

PROCEDURE Einrichten(VAR B : TREE; kl : KleinerProzedur);
    (*
     * Richtet einen binären Baum mit Rückzeiger ein,
     * der gemäß 'kl' geordnet ist.
     *)

PROCEDURE leer(B : TREE): BOOLEAN;
    (*
     * Ist 'TRUE', wenn der Baum leer ist.
     *)

PROCEDURE gefunden(B : TREE): BOOLEAN;
    (*
     * Diese Prozedur ist nach Aufruf von 'Erster', 'Voriger', 'Naechster'
     * und 'Suche' anzuwenden. Sie liefert 'TRUE', wenn nach diesen Proze-
     * duren im Baum etwas gefunden wird.
     *)

PROCEDURE Erster(B : TREE);
    (*
     * Sucht nach dem kleinsten Element im Baum.
     *)

PROCEDURE Naechster(B : TREE);
    (*
     * Sucht das nächstgrößte Element im Baum.
     *)

PROCEDURE Voriger(B : TREE);
    (*
     * Sucht das nächstkleinere Element im Baum.
     *)

PROCEDURE Suchen(B : TREE; ziel : ARRAY OF BYTE);
    (*
     * Sucht nach einem Element mit dem Schlüssel 'ziel' im Baum.
     *)
```

```
PROCEDURE Einfuegen(B : TREE; inhalt: ARRAY OF BYTE);
       (*
        *  Fügt das Datum 'inhalt' in den Baum.
        *)

PROCEDURE  Loeschen(B : TREE);
       (*
        *  Ein zu löschendes Element sucht man  zunächst mit 'Suchen'.
        *  Ergibt anschließend 'gefunden' TRUE, so kann man es mit
        *  'Loeschen' aus dem Baum entfernen.
        *)

PROCEDURE HoleDaten(B : TREE; VAR inhalt: ARRAY OF BYTE);
       (*
        *  Man sucht einen Schlüssel zunächst mit 'Suchen'. Ergibt 'gefunden'
        *  anschließend 'TRUE', so können die abgespeicherten Daten mit
        *  'HoleDaten' ausgelesen werden.
        *)

PROCEDURE GebeDaten(B: TREE; VAR inhalt: ARRAY OF BYTE);
       (*
        *  Diese Prozedur sollte nur im Zusammenhang mit 'Lesen' in
        *  'LinearZuBaum' genutzt werden (s.u).
        *)

PROCEDURE BaumZuLinear(B : TREE; Schreiben : PROC);
       (*
        *  Überführt eine Baumstruktur in eine lineare Struktur (ARRAY, File).
        *  Wie das im einzelnen geschieht, wird in 'Schreiben' festgelegt.
        *  Die '<' - Relation wird automatisch  dabei berücksichtigt.
        *  Typische Anwendung: Sortiertes Abspeichern der Bauminhalte auf eine
        *  Datei: 'Schreiben' ruft 'HoleDaten' auf und schiebt jedes Datum auf
        *  die Datei.
        *)

PROCEDURE LinearZuBaum(ElementZahl : CARDINAL;
                       Lesen       : PROC;
                       kleiner     : KleinerProzedur;
                       VAR B       : TREE);
       (*
        *  Überführt eine lineare, gemäß 'kleiner' sortierte Struktur in einen
        *  ausgeglichen Baum.
        *  Wie das im einzelnen geschieht, wird in 'Lesen' festgelegt.
        *  'Lesen' wird sequentiell von 'LinearZuBaum' aufgerufen und
```

```
    * muß den nächsten Datensatz mit 'GebeDaten' dem Baum Übergeben.
    * 'Elementzahl' ist die Anzahl der zu speichernden Daten.
    * Typische Anwendung: Einlesen der mit 'BaumZuLinear' auf einer Datei
    * abgespeicherten Bauminhalte.
    *)

END Baum.
```

Der Implementationsmodul »Baum«

Hier ein Rat an den Leser: Wie erwähnt, ist die Implementation eines Baumes mit Rückzeiger, der auf Daten beliebigen Typs operiert, nicht ganz einfach. Wenn Sie vor lauter Zeigern nicht mehr sehen, wo es lang geht, so überschlagen Sie diesen Textabschnitt beim ersten Lesen. Um mit Bäumen umzugehen, ist es nicht wichtig zu wissen, wie sie implementiert sind (Geheimnisprinzip). Lesen Sie dazu den nächsten Teilabschnitt und lassen Sie das Demoprogramm laufen.

Hier also nun weiter für die hartgesottenen »Zeigerverbieger».

Zunächst einige Begriffserklärungen für Bäume:

Knoten:
Verbundvariable mit Inhalt und je einem Zeiger auf den nächsten linken und rechten Knoten. Im allgemeinen enthält der linke Knoten dabei einen kleineren Inhalt, der rechte einen größeren als der Knoten selbst. Oft sind auch Rückzeiger zum Vorgänger vorhanden.

Wurzel:
Oberster Knoten. Der Rückzeiger der Wurzel ist `NIL`.

Blatt (Endknoten):
Ein unterer Knoten, dessen beide Zeiger links und rechts `NIL` sind.

Teilbaum:
Baum, dessen Wurzel ein Knoten eines anderen Baumes ist.

Ein nicht leerer binärer Baum besteht also immer aus einer Wurzel und wenigstens einem Teilbaum. Diese recht einfach formulierte Weisheit enthält eine wesentliche Strategie für effiziente Prozeduren zur Verwaltung von Bäumen: rekursive Programmierung.

Alle Daten eines Baumes zu bearbeiten (oder auszugeben) ist hiermit ganz einfach:

```
PROCEDURE bearbeiten(p: KnotenPtr);
BEGIN
  IF p # NIL THEN
      bearbeiten(p^.links);
      <bearbeiten oder drucken von p^.info>
      bearbeiten(p^.rechst)
  END
END bearbeiten;
```

Mit `bearbeiten(wurzel)` kann man so zum Beispiel den gesamten Baum ausgeben; die Prozedur ist (fast) so einfach wie eine FOR-Schleife bei einem Feld:

```
FOR i : = MIN(bereich) TO MAX (bereich) DO
    <bearbeiten oder drucken von feld[i]>;
END;
```

Hat man diese grundlegenden Verfahren einmal richtig durchschaut, ist das Verständnis für die folgenden Prozeduren und Funktionen einfach zu erlangen.

Doch zunächst einmal zur Datenstruktur:

Wir exportieren den Datentyp `TREEopak`. Ein Zeiger diese Typs zeigt auf einen Verbund vom Typ `TREEHeader`; dieser besteht aus der Wurzel, einem Zeiger `Aktueller` (zeigt auf den gerade zu bearbeitenden Knoten) und einer Prozedur-Variablen `kleiner`. Nach der dort stehenden Prozedur wird der Baum sortiert.

Mit `Aktueller` und `Wurzel` haben wir Zugriff auf die Knoten des Baumes, in denen wir wieder einen Zeiger `Eintrag` finden, der auf die eigentlichen Daten zeigt:

Im Implementationsmodul gibt es wieder Testprozeduren, die bei der Programmentwicklung hilfreich waren: `CheckKnoten` (Konsistenz-Überprüfung eines Knotens) und `CheckTree` (Überprüfung der korrekten Verzeigerung der gesamten Baumstruktur).

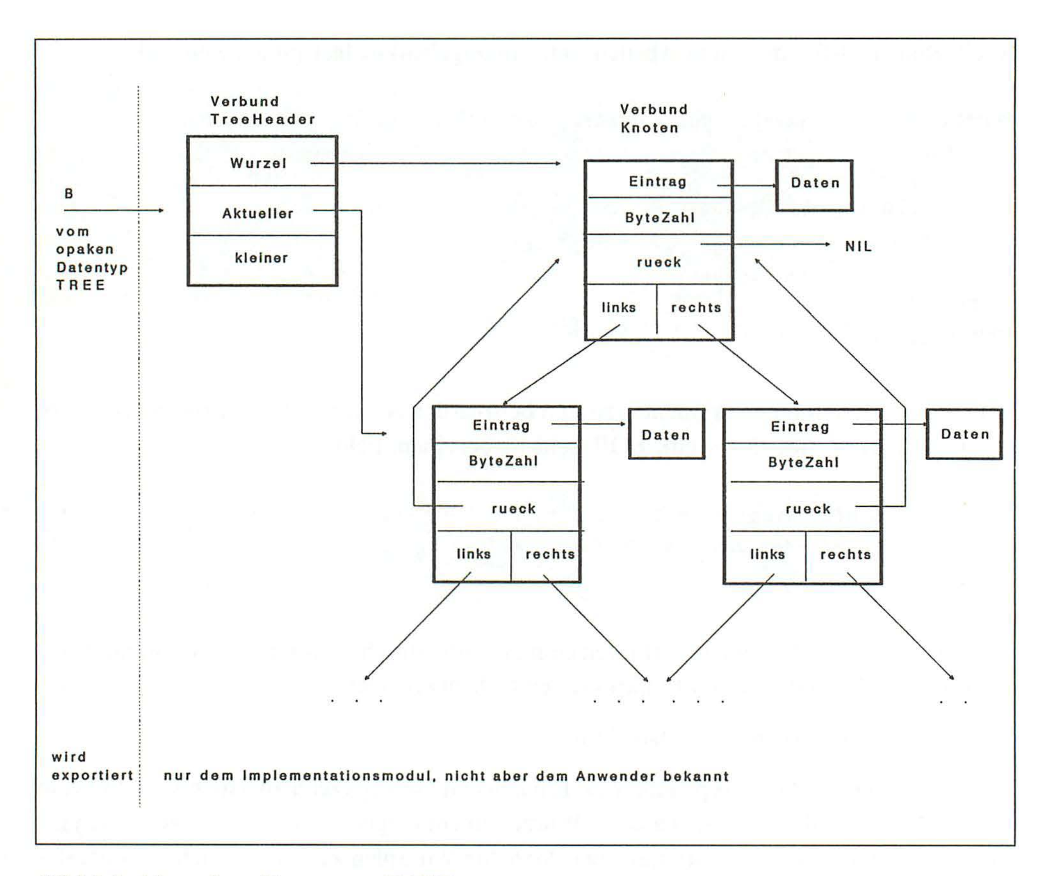


Bild 2.8: Abstrakter Datentyp »TREE«

Im Implementationsmodul folgen einige Hilfsprozeduren:

- `CopyN` zum Kopieren zweier Variablen beliebigen Typs (ist bereits bekannt).
- `NeuerKnoten` erzeugt einen neuen Knoten.
- `LöscheKnoten` löscht den Knoten und gibt den Speicherplatz wieder frei.

Nun zu den wichtigen exportierten Prozeduren:

`Einfügen` zeigt eine rekursive Vorgehensweise: Man sucht die Einfügestelle indem man von der Wurzel ausgehend den Baum hinabsteigt. Ist der einzufügende Inhalt kleiner als der gerade betrachtete Inhalt, so ist im linken Teilbaum einzufügen, sonst im rechten.

Neue Knoten werden grundsätzlich als Blätter eingefügt, ihr linker und rechter Teilbaum sind also `NIL`. Das hat den Vorteil, daß man nicht an bestehenden Verkettungen anderer Knoten herumbiegen muß.

Das Einfügen von Inhalten mit gleichem Schlüssel ist nach unserer Prozedur zulässig! Wenn man zum Beispiel einen Verbund von Nachnamen und Vornamen einer Person im Baum verwaltet, und die »kleiner-Prozedur« nur die Nachnamen vergleicht, so wird ein neuer »Meier« stets rechts vom bisherigen »Meier« eingefügt. Der neue »Meier« ist also »größer« als der alte.

Das Löschen in einem Baum ist die komplizierteste Prozedur, sie demonstriert das »Zeigerverbiegen« in Reinkultur.

Nehmen wir an, wir haben den Knoten gefunden, den wir löschen wollen (andernfalls brauchen wir ja nichts zu löschen). Nennen wir ihn `K`.

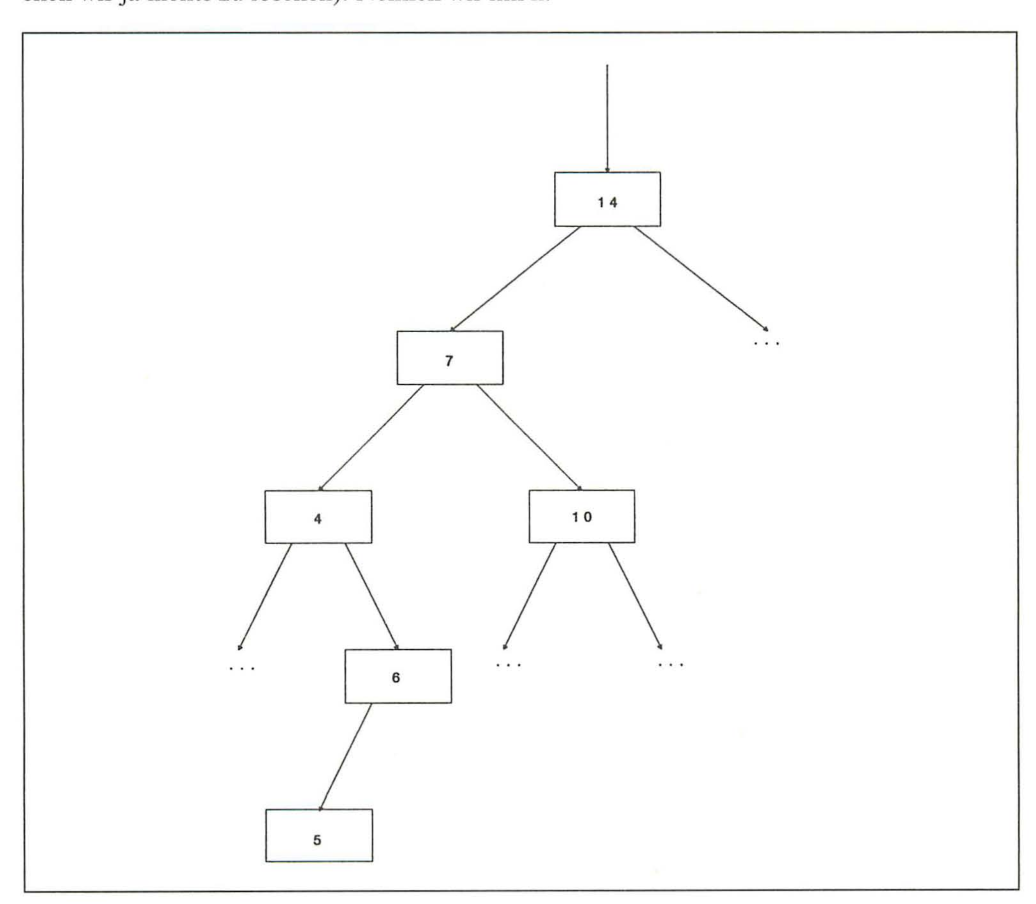


Bild 2.9: »Baum vor dem Löschen«

Dann unterscheiden wir zwei Fälle:

1. `K` besitzt höchstens einen nicht leeren Teilbaum; also `K.links` oder `K.rechts` ist `NIL`
2. kein Teilbaum ist leer, also `K.links#NIL` und `K.rechts#NIL`

Zu 1.
Hier können wir `K` einfach »ausketten«, indem man den Vorgänger von `K` mit seinem Teilbaum verkettet: Nehmen wir an, `K.links` sei `NIL`. Dann biegt man den Pointer des Vorgängers, der auf `K` zeigt, um auf `K^.rechts`.

Zu 2.
Weil `K` zwei nicht leere Teilbäume besitzt, können wir den Vorgänger nicht einfach auf einen der Teilbäume durchketten, da der andere sonst »in der Luft« hinge. Wenn wir `K` allerdings einfach durch seinen nächst kleineren ersetzten (den wir `J` nennen), stimmt die Reihen-

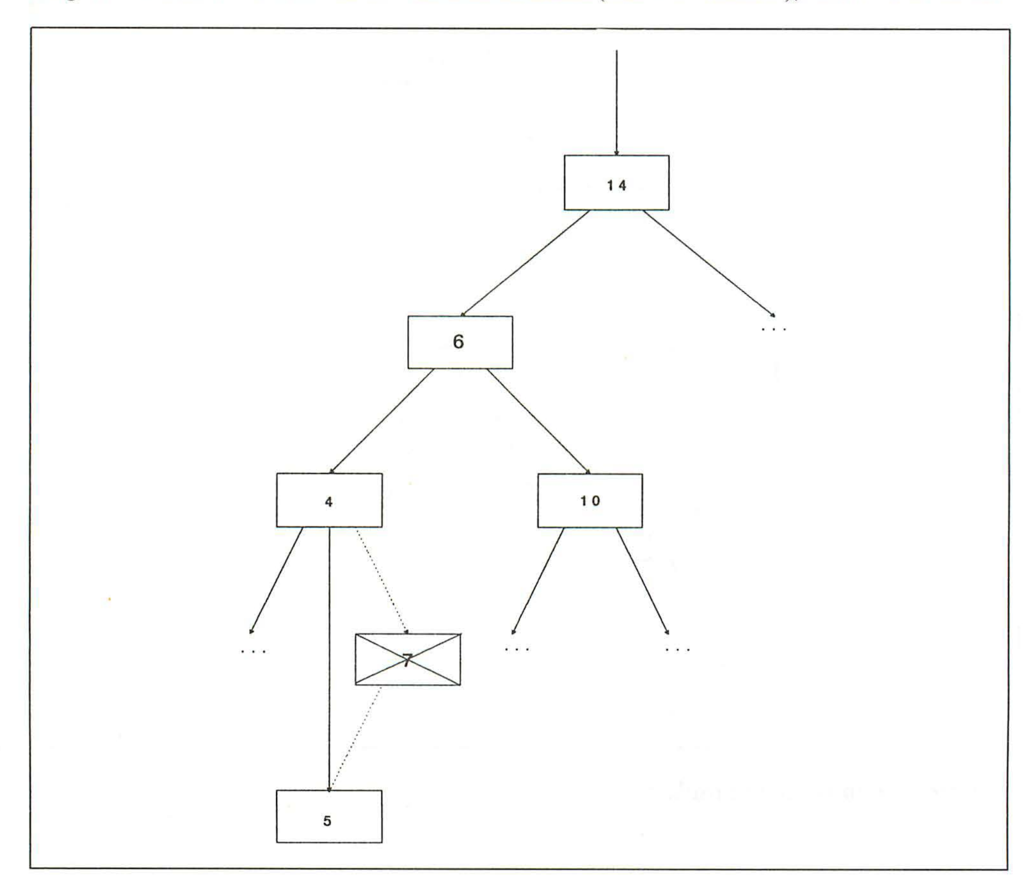


Bild 2.10: Baum nach dem Löschen des Knotens »7«

folge im Baum immer noch. `J` ist der größte im linken Teilbaum von `K und J.links` ist `NIL` (sonst wäre dieser ja größer als `J`); wir können ihn also einfach wie bei 1. ausketten, und für `K` einketten.

Da wir das »Überketten« eines Knotens (also das Verbinden seines Vorgängers mit einem seiner Nachfolger) öfter brauchen, erledigt das die Prozedur `linken` (»verbinden«). `linke(a, b, c)` kettet also `b` aus der Kette `a->b->c` aus, indem es `a->c` verbindet.

Die Implementation der Prozedur `HoleDaten`, `Erster` und `Letzter` ist selbsterklärend. Etwas kniffliger ist `Naechster`, das den Zeiger `Aktueller` auf den nächst größeren Knoten setzt. Hier gibt es drei Fälle zu unterscheiden:

- Wenn `Aktueller=NIL` ist, gibt es keinen größeren Knoten.
- Wenn `Aktueller. rechts=NIL` ist, geht man solange mittels des Rückzeigers im Baum zurück, bis der Rückzeiger nach rechts führt. Landet man vorher bei der Wurzel, gibt es keinen größeren mehr.
- Ansonsten ist `Aktueller. rechts` ungleich `NIL` und es gibt einen rechten Teilbaum. Von dem nimmt man das kleinste Element: man geht also solange nach links, bis es dort keinen »linkeren« (kleineren) mehr gibt.

Die Prozedur `Voriger` läuft analog.

```
IMPLEMENTATION MODULE Baum;

FROM SYSTEM  IMPORT ADR, ADDRESS, BYTE, TSIZE;
FROM Storage IMPORT ALLOCATE, DEALLOCATE, Available;

CONST
    Magic = 12345;       (* nur zum Check *)

TYPE
     TREE       = POINTER TO TreeHeader;
     KnotenPtr  = POINTER TO Knoten;
     TreeHeader = RECORD
                     Wurzel, Aktueller : KnotenPtr;
                     kleiner           : KleinerProzedur;
                   END;
     Knoten     = RECORD
                    ByteZahl              : CARDINAL;
                    Eintrag               : ADDRESS;
                    links, rechts, rueck : KnotenPtr;
                  END;
```

```
(* --------- Die folgende Prozeduren dienen nur zu Testzwecken ---------- *)
PROCEDURE checkKnoten(b: TREE; p: KnotenPtr);
BEGIN                                   (* Test auf korrekte Verzeigerung *)
  WITH p^ DO
    IF rueck = NIL THEN
      IF p # b^.Wurzel THEN HALT END
      ELSE
        IF (rueck^.links # p) AND (rueck^.rechts # p) THEN HALT END
    END;
    IF (links  # NIL) AND (links ^.rueck # p) THEN HALT END;
    IF (rechts # NIL) AND (rechts^.rueck # p) THEN HALT END
  END
END checkKnoten;

PROCEDURE checkTree(B: TREE);           (* Test auf korrekte Verzeigerung *)
    PROCEDURE ct(p, vor: KnotenPtr);
    BEGIN
      IF p <> NIL THEN
        IF p^.rueck <> vor THEN HALT END;
        ct(p^.links, p);
        ct(p^.rechts, p) END
    END ct;
BEGIN   (* checkTree *)
  ct(B^.Wurzel, NIL)
END checkTree;

(* ------ Ende Testprozeduren, Beginn des eigentlichen Impl. Moduls ------- *)

(* ------------------------ Hilfsprozeduren ----------------------------- *)
PROCEDURE CopyN(von, nach: ADDRESS; groesse: CARDINAL);
VAR
    i: CARDINAL;
    pNach, pVon: POINTER TO BYTE;
BEGIN
  pVon := von;
  pNach := nach;
  FOR i := 1 TO groesse DO
    pNach^ := pVon^;
    INC(pNach); INC(pVon)
  END
END CopyN;

PROCEDURE NeuerKnoten(VAR p: KnotenPtr; groesse: CARDINAL);
BEGIN
```

```
  ALLOCATE(p, TSIZE(Knoten));
  WITH p^ DO
    rueck := NIL; links := NIL; rechts := NIL;
    ALLOCATE(Eintrag, LONG(groesse))
  END
END NeuerKnoten;

PROCEDURE LoescheKnoten(VAR p: KnotenPtr);
BEGIN
  DEALLOCATE(p^.Eintrag, p^.ByteZahl);
  DEALLOCATE(p, TSIZE(Knoten));
  p := NIL
END LoescheKnoten;

(* -- Ende Hifsprozeduren, Beginn der Impl. der exportierten Prozeduren --- *)
PROCEDURE Einrichten(VAR B: TREE; kl: KleinerProzedur);
BEGIN
  ALLOCATE(B, TSIZE(TreeHeader));
  WITH B^ DO
    Wurzel    := NIL; Aktueller := NIL; kleiner := kl
  END
END Einrichten;

PROCEDURE leer(B: TREE) : BOOLEAN;
BEGIN
  RETURN B^.Wurzel = NIL
END leer;

PROCEDURE gefunden(B: TREE): BOOLEAN;
BEGIN
  RETURN B^.Aktueller # NIL
END gefunden;

PROCEDURE Einfuegen(B : TREE; inhalt : ARRAY OF BYTE);

    PROCEDURE Einfuegenl(last: KnotenPtr; VAR p: KnotenPtr);(* lok. Hilfspr *)
    BEGIN
      IF p = NIL THEN
        NeuerKnoten(p, HIGH(inhalt)+1);
        WITH p^ DO
          rueck := last; links := NIL;  rechts := NIL;
          ByteZahl := HIGH(inhalt)+1;
          CopyN(ADR(inhalt), Eintrag, HIGH(inhalt)+1)
```

```
        END;
        checkKnoten(B, p);
        B^.Aktueller := p
      ELSE
        checkKnoten(B, p);
        IF B^.kleiner(ADR(inhalt), p^.Eintrag) THEN
          Einfuegenl(p, p^.links)
        ELSE
          Einfuegenl(p, p^.rechts)
        END
      END
    END Einfuegenl;                              (* Ende lok. Hilfsproz.*)

BEGIN   (* Einfuegen *)
  Einfuegenl(NIL, B^.Wurzel)
END Einfuegen;

PROCEDURE Loeschen(B: TREE);
VAR p: KnotenPtr;
    a: ADRESS;
    c: CARDINAL;

  PROCEDURE linken(vor,alt,auf: KnotenPtr);    (* lokale Hilfsproz. *)
  BEGIN
    checkKnoten(B,alt);
    IF vor = NIL THEN B^.Wurzel := auf
    ELSIF vor^.rechts = alt THEN vor^.rechts := auf
    ELSIF vor^.links  = alt THEN vor^.links  := auf
    ELSE HALT END;
    IF auf # NIL THEN checkKnoten(B,auf); auf^.rueck := vor END
  END linken;                                  (* Ende lokale Hilsproz. *)

BEGIN   (* Loeschen *)
  WITH B^ DO
    IF Aktueller = NIL THEN HALT END;
    WITH Aktueller^ DO
      IF rechts = NIL THEN
        linken(rueck, Aktueller, links)
      ELSIF links = NIL THEN
        linken(rueck, Aktueller, rechts)
      ELSE          (*'Aktueller' mit seinem nächst kleineren vertauschen *)
        p := links;
        WHILE p^.rechts # NIL DO p := p^.rechts END;
        linken(p^.rueck, p, p^.links);  (*Blatt 'p' aus alter Pos ausketten *)
```

```
        a := Eintrag; := p^.Eintrag := a;
        c := ByteZahl; Bytezahl := p^.ByteZahl p^.Bytezahl := c;
        Aktueller := p;            (* 'p' ist jetzt ungültig und zu löschen *)
      END
    END;
    LoescheKnoten(Aktueller);
  END
END Loeschen;

PROCEDURE HoleDaten(B: TREE; VAR inhalt: ARRAY OF BYTE);
BEGIN
  IF B^.Aktueller = NIL THEN HALT END;          (* Programmierfehler, es ... *)
  WITH B^.Aktueller^ DO                          (* ... ist nichts zu Holen.  *)
    IF ByteZahl # HIGH(inhalt) + 1 THEN HALT END;
    CopyN(Eintrag, ADR(inhalt), ByteZahl)
  END
END HoleDaten;

PROCEDURE GebeDaten(B: TREE; VAR inhalt: ARRAY OF BYTE);
BEGIN
  IF B^.Aktueller = NIL THEN HALT END;
  WITH B^.Aktueller^ DO
    IF Eintrag <> NIL THEN HALT END;      (* Test: 'GebeDaten' darf nur im... *)
    IF ByteZahl <> Magic THEN HALT END; (* ...Zusammenhang mit LinearZuBaum *)
    ByteZahl := HIGH(inhalt)+1;                  (* ...aufgerufen werden   *)
    ALLOCATE(Eintrag, ByteZahl);
    CopyN(ADR(inhalt), Eintrag, ByteZahl)
  END
END GebeDaten;

PROCEDURE Erster(B: TREE);
BEGIN
  WITH B^ DO
    Aktueller := Wurzel;
    IF Aktueller = NIL THEN RETURN END;
    WHILE Aktueller^.links # NIL DO
      Aktueller := Aktueller^.links END
  END
END Erster;

PROCEDURE Naechster(B: TREE);
VAR letzter: KnotenPtr;
BEGIN
  WITH B^ DO
```

```
    IF Aktueller = NIL THEN RETURN END;
    IF Aktueller^.rechts = NIL THEN
      REPEAT
        letzter := Aktueller;
        Aktueller := Aktueller^.rueck
      UNTIL (Aktueller = NIL) OR (Aktueller^.links = letzter);
    ELSE
      Aktueller := Aktueller^.rechts;
      WHILE Aktueller^.links # NIL DO Aktueller := Aktueller^.links END
    END
  END
END Naechster;

PROCEDURE Voriger(B: TREE);
VAR letzter: KnotenPtr;
BEGIN
  WITH B^ DO
    IF Aktueller = NIL THEN RETURN END;
    IF Aktueller^.links = NIL THEN
      REPEAT
        letzter := Aktueller;
        Aktueller := Aktueller^.rueck
      UNTIL (Aktueller = NIL) OR (Aktueller^.rechts = letzter);
    ELSE
      Aktueller := Aktueller^.links;
      WHILE Aktueller^.rechts # NIL DO Aktueller := Aktueller^.rechts END
    END
  END
END Voriger;

PROCEDURE Suchen(B: TREE; ziel: ARRAY OF BYTE);
VAR p: KnotenPtr;
BEGIN
  B^.Aktueller := NIL;
  p := B^.Wurzel;
  WHILE p # NIL DO
    B^.Aktueller := p;
    IF B^.kleiner(p^.Eintrag, ADR(ziel))
      THEN p := p^.rechts
      ELSE p := p^.links
    END
  END;
  IF (B^.Aktueller # NIL) AND B^.kleiner(B^.Aktueller^.Eintrag,ADR(ziel)) THEN
    Naechster(B)
```

```
  END
END Suchen;

PROCEDURE BaumZuLinear(B : TREE; Schreiben : PROC);

    PROCEDURE BzL(kp: KnotenPtr);                 (* lokale Hilfsprozedur *)
    BEGIN
      IF kp # NIL THEN
        BzL(kp^.links);
        B^.Aktueller := kp;
        Schreiben;      (* ruft zum Schreiben eines Datums 'HoleDaten' auf *)
        BzL(kp^.rechts)
      END
    END BzL;                                      (* Ende lok. Hilfsproz. *)

BEGIN  (* BaumZulinear *)
  BzL(B^.Wurzel);
  B^.Aktueller := NIL;
END BaumZuLinear;

PROCEDURE LinearZuBaum(
                  ElementZahl: CARDINAL;
                  Lesen      : PROC;
                  kleiner    : KleinerProzedur;
                  VAR B      : TREE);
VAR datenPtr : ADDRESS;

    PROCEDURE LzB(zahl: CARDINAL; vor: KnotenPtr): KnotenPtr;  (* lok.Hilfp *)
    VAR neu: KnotenPtr;
    BEGIN
      IF zahl = 0 THEN neu :=  NIL
      ELSE
        ALLOCATE(neu, TSIZE(Knoten));
        neu^.rueck := vor;
        neu^.links := LzB(zahl DIV 2, neu);
        neu^.Eintrag := NIL;        (* Zur Sicherheit: wird von 'GebeDaten   *)
        neu^.ByteZahl := Magic;     (* ...abgeprüft *)
        B^.Aktueller := neu;
        Lesen;               (* ruft zum Lesen eines Datums 'Gebedaten' auf *)
        neu^.rechts := LzB(zahl - 1 - zahl DIV 2, neu)
      END;
      RETURN neu;
    END LzB;                                    (* Ende lokale Hilfsproz. *)
```

```
BEGIN  (* LinearZuBaum *)
  Einrichten(B, kleiner);
  B^.Wurzel := LzB(ElementZahl, NIL);
  B^.Aktueller := NIL;
  checkTree(B)
END LinearZuBaum;
END Baum.
```

Das folgende Programm demonstriert das Einfügen, Löschen und Suchen sowie das »Blättern« mit den Prozeduren `Voriger` und `Naechster`.

```
MODULE BaumDemo;

FROM SYSTEM IMPORT ADDRESS, ADR;
FROM InOut  IMPORT WritePg, WriteLn, Write, WriteString, WriteLHex,
                   Read, ReadString, ReadCard;
FROM Baum   IMPORT TREE, KleinerProzedur, Einrichten, Einfuegen,
                   Loeschen, leer, Suchen, gefunden,
                   Erster, Naechster, Voriger, HoleDaten, GebeDaten,
                   LinearZuBaum, BaumZuLinear;

IMPORT Strings;

TYPE
    Str10  = ARRAY[0..9] OF CHAR;
    Person = RECORD
               Name, Vorname : Str10
             END;

VAR B : TREE;

PROCEDURE kleiner(a1,a2 : ADDRESS) : BOOLEAN;
VAR p1,p2: POINTER TO Person;
BEGIN
  p1 := a1; p2 := a2;
  RETURN Strings.Compare(p1^.Name,p2^.Name) = Strings.less
END kleiner;

PROCEDURE Taste;
VAR ch : CHAR;
BEGIN
  WriteLn; WriteString("<Bitte Taste drücken>"); Read(ch)
END Taste;
```

```
PROCEDURE Schreibe(VAR p : Person);
VAR c: CHAR; i: CARDINAL;
BEGIN
  WriteLn;
  WriteLn; WriteString("Name...: "); WriteString(p.Name);
  WriteLn; WriteString("Vorname: "); WriteString(p.Vorname);
  WriteLn
END Schreibe;

PROCEDURE ListeDrucken(B : TREE);
VAR
    daten : Person;
BEGIN
  WritePg;                                        (* Bildschirm loeschen *)
  Erster(B);                            (* Sucht den kleinsten Datensatz *)
  WHILE gefunden(B) DO
    HoleDaten(B, daten);
    WriteLn; WriteString(daten.Name);
    WriteString(", "); WriteString(daten.Vorname);
    Naechster(B)
  END;
  Taste
END ListeDrucken;

PROCEDURE Anzahl : CARDINAL;                (* Hilfsproz. für LinearZuBaum *)
VAR anz: CARDINAL;
BEGIN
  WriteLn; WriteString("Anzahl: "); ReadCard(anz); RETURN anz
END Anzahl;

PROCEDURE Schreiben;          (* als Ausgabedatei wird der Bildschirm benutzt *)
VAR datum : Person;
BEGIN
  HoleDaten(B,datum);
  WriteLn; WriteString("Name: ");    WriteString(datum.Name);
           WriteString(", Vorname: "); WriteString(datum.Vorname);
END Schreiben;

PROCEDURE Lesen;                (* Als Eingabedatei wird die Tastatur benutzt *)
VAR datum: Person;
BEGIN
  WriteLn; WriteString("Name    : "); ReadString(datum.Name);
           WriteString("Vorname : "); ReadString(datum.Vorname);
  GebeDaten(B,datum)
```

```
END Lesen;

VAR
    wahl, taste : CHAR;
    i           : CARDINAL;
    data        : Person;

BEGIN
Einrichten(B, kleiner);
REPEAT
    WritePg;                                        (* Bildschirm loeschen *)
    IF gefunden(B) THEN
       HoleDaten(B, data);
       Schreibe(data)
    ELSE
       WriteLn; WriteString(" *** Kein Datensatz gewaehlt ***")
     END;
    WriteLn;
    WriteLn; WriteString("E(infügen, S(uchen, D(rucken"); WriteLn;
    WriteString("A(uf Datei(=Bildschirm), H(olen von Datei(=Tastatur)");
    IF gefunden(B) THEN
      WriteLn; WriteString("V(origer, N(ächster, L(oeschen")
    END;
    WriteLn; WriteString("<ESC> = Ende");
    WriteLn; Read(wahl); WriteLn;
    CASE CAP(wahl) OF
      "N" : Naechster(B) |
      "V" : Voriger(B)   |
      "E" : WriteLn; WriteString("Nachname: "); ReadString(data.Name);
            WriteLn; WriteString("Vorname:  "); ReadString(data.Vorname);
            Einfuegen(B, data) |
      "L" : IF gefunden(B) THEN Loeschen(B) END |
      "S" : WriteLn; WriteString("Welchen Namen Suchen: ");
            ReadString(data.Name);
            Suchen(B, data) |
      "D" : ListeDrucken(B) |
      "A" : BaumZuLinear(B,Schreiben); Taste |
      "H" : WriteLn; WriteString("Namen in alphabet. Reihenfolge eingeben!!!");
            LinearZuBaum(Anzahl(),Lesen,kleiner,B)
    END
UNTIL wahl = 33C
END BaumDemo.
```

Sie werden an diesem Beispielprogramm bemerkt haben, daß der Baum nicht immer »ausgeglichen« ist, das heißt, daß die Tiefe des linken und rechten Teilbaumes eines Knoten differiert. Im extremen Fall kann ein Baum so zu einer »linearen Liste« entarten, zum Beispiel wenn nacheinander in der Größe aufsteigende Elemente eingefügt werden. Hier wäre dann der linke Teilbaum der Wurzel leer.

In einem solchen Fall verlängern sich selbstverständlich die Zugriffszeiten beim Suchen. Die ideale binäre Struktur ist ja eine Vorraussetzung für ein Suchen mit wenigen Zugriffen (und Vergleichen). Hier verschafft folgendes Vorgehen Abhilfe: Man führt einen »Baumausgleich« durch. Das geschieht automatisch, wenn man den Baum mit `BaumZuLinear` auf Diskette schreibt und ihn wieder mit `LinearZuBaum` einliest. Das Ergebnis ist ein ausgeglichener Baum.

Eine andere Methode ist es, direkt beim Einfügen für einen ausgeglichenen Baum zu sorgen. Diese Idee wurde zum erstenmal von den russischen Informatikern *Adelson-Velskii* und *Landis* implementiert. Man spricht von AVL-Bäumen. Hier ist das Maß für die Nichtsymmetrie die Höhendifferenz der beiden Unterbäume. In einem ausgeglichenen Baum kann diese Differenz nur 0, 1 oder -1 betragen. Sollte sich hieran beim Einfügen oder Löschen etwas ändern, so wird nach dem »Mobile-Prinzip« rekonfiguriert. Bildlich gesprochen: Hängt das Mobile schief (Höhendifferenz betragsmäßig > 1), geht man einen Knoten nach links bzw. rechts und hängt es daran wieder auf. Die Implementation eines AVL-Baumes mit Rückzeigern würden den Rahmen diese Buches sprengen. Wir verweisen auf die weiterführende Literatur [W6]. Nicht verschwiegen werden soll auch der Geschwindigkeitsverlust beim Einfügen und Löschen, der mit dem dauernden Ausgleichen verbunden ist. Eine realistische Anwendung unseres Baummoduls zeigt der nächste Abschnitt über Dateiverwaltung.

Programmieren lernt man nur durch programmieren! Deshalb zum Schluß noch eine Anregung, wenn Sie selbst mit der Implementation von verketteten Datenstrukturen Erfahrungen sammeln wollen: Schreiben Sie einen externen Modul für doppelt verkettete lineare Listen (vgl. Abb. »Zeigerstrukturen« aus Abschnitt 1.6.6). Sie können natürlich die gleichen Prozedurnamen mit den gleichen Funktionen wie im Baummodul implementieren. Es ist nur viel einfacher, da sie es mit einer linearen Struktur zu tun haben und Vorgänger bzw. Nachfolger immer das nächst kleinere oder größere Element beinhalten.

2.2.4 Software-Engineering bei verzeigerten Strukturen

Wer viel arbeitet, macht auch viele Fehler! Dieser Spruch gilt insbesondere für die Programmierarbeit mit verzeigerten Strukturen. Wie man an den vorangegangenen Beispielen erkennen konnte, haben wir beim Schreiben von Implementationsmodulen stets einige Testprozeduren mitaufgeführt, die zu dem Zeitpunkt der Programmerstellung die Korrektheit der Ver-

zeigerung nach jeder Operation prüften. Aus didaktischen Gründen haben wir diese Überprüfungsroutinen nicht gelöscht, sie mögen als Anschauungsmaterial gelten.

Ein wesentlicher Zweck der Testprozeduren war die Terminierung von Programmen; hierzu wurde des öfteren die Prozedur `HALT` aufgerufen, wenn etwas nicht stimmte. Mit einem Debugger läßt sich so eine fehlerhafte Verzeigerung zurückverfolgen. Zur genaueren Überprüfung des momentanen Heap-Inhaltes dient auch der Modul `TeStorage`, der in diesem Abschnitt vorgestellt werden soll. Er stellt die Prozedur `Speicherliste` zur Verfügung, die einen »Dump« auf den Bildschirm erledigt. Dabei werden die Speicherinhalte des Heaps mit ihrer Bytezahl ausgegeben, numeriert in der Reihenfolge wie sie angefordert wurden. Außerdem enthält er noch die Prozeduren `ALLOCATE` und `DEALLOCATE`, die hier mit gleichem Namen wie in `Storage` vorkommen, jedoch leistungsfähiger sind, da die Ausgabe der Speicherliste mit vorbereitet wird. Der Knüller dieser Namensgebung liegt darin, daß nach der Testphase des zu entwickelnden Programms in der Importliste die Buchstaben »`Te`« bei »`TeStorage`« gelöscht werden können, und alles läuft wieder unter `Storage` ab. Selbstverständ lich sind zuvor die Aufrufe von `Speicherliste` aus dem fehlerfreien Programm zu entfernen.

```
DEFINITION MODULE TeStorage;
        (*
         * Dieser Modul ist in der Entwicklungsphase bei Modulen statt
         * 'Storage' zu nutzen. Er ermöglicht nach jeder Zeigeroperation
         * die Ausgabe einer Liste der Speicherinhalte auf dem Heap.
         * Ist ein lauffähiges Programm erstellt, so ersetzt man in der
         * IMPORT-Liste einfach 'TeStorage' durch 'Storage'!
         *)

FROM SYSTEM IMPORT ADDRESS;

TYPE
     SizeType = LONGCARD;

PROCEDURE ALLOCATE(VAR ptr: ADDRESS; groesse: SizeType);
PROCEDURE DEALLOCATE(VAR ptr: ADDRESS; groesse: SizeType);
PROCEDURE SpeicherListe;

END TeStorage.
```

```
IMPLEMENTATION MODULE TeStorage;

FROM SYSTEM IMPORT TSIZE, ADDRESS;
FROM InOut  IMPORT Write, WriteLn, WriteString, WriteCard;
IMPORT Storage;
```

```
CONST
    magCon1 = 12345;
    magCon2 = 31415;

TYPE
    nodePtr = POINTER TO node;
    node    = RECORD
                  magic1 : CARDINAL;
                  link   : nodePtr;
                  count  : CARDINAL;
                  size   : SizeType;
                  data   : ADDRESS;
                  magic2 : CARDINAL
              END;
VAR
    MasterCount : CARDINAL;
    StoreList   : nodePtr;

PROCEDURE Check(p: nodePtr);
BEGIN
  IF p^.magic1 <> magCon1 THEN HALT END;
  IF p^.magic2 <> magCon2 THEN HALT END
END Check;

PROCEDURE MakeCheck(p: nodePtr);
BEGIN
  p^.magic1 := magCon1;
  p^.magic2 := magCon2;
END MakeCheck;

PROCEDURE ALLOCATE(VAR ptr : ADDRESS; groesse : SizeType);
VAR
    n : nodePtr;
BEGIN
    INC(MasterCount);
    WriteString(" <ALLOC("); WriteCard(MasterCount,1); Write(",");
    WriteCard(groesse,1); WriteString(")> ");
    Storage.ALLOCATE(n, TSIZE(node));
    IF n = NIL THEN HALT END;
    Storage.ALLOCATE(ptr, groesse);
    IF ptr = NIL THEN HALT END;      (* <!> kein freier Speicher *)
    n^.link := StoreList;
```

```
    n^.count := MasterCount;
    n^.size := groesse;
    n^.data := ptr;
    MakeCheck(n);
    StoreList := n;
END ALLOCATE;

PROCEDURE DEALLOCATE(VAR ptr : ADDRESS; groesse : SizeType);
VAR
    vor, n : nodePtr;
BEGIN
    vor := NIL;
    n := StoreList;
    LOOP
      IF n = NIL THEN HALT END;                    (* <!> nicht alloziert *)
      Check(n);
      IF n^.data = ptr THEN EXIT END;
      vor := n;
      n := n^.link
    END;
    WriteString(" <DEALL("); WriteCard(n^.count,1);Write(",");
    WriteCard(n^.size,1); WriteString(")> ");
    IF n^.size <> groesse THEN HALT END;           (* <!> falsche Groesse *)
    IF vor = NIL THEN
        StoreList := n^.link
    ELSE
        vor^.link := n^.link
    END;
    Storage.DEALLOCATE(n^.data, n^.size);
    Storage.DEALLOCATE(n, TSIZE(node))
END DEALLOCATE;

PROCEDURE SpeicherListe;
VAR p: nodePtr;
BEGIN
    WriteLn; WriteString("===  Noch im Heap:   ===");
    p := StoreList;
    WHILE p <> NIL DO
      Check(p);
      WriteLn;
      WriteString("-- ");
      WriteCard(p^.count,6); WriteCard(p^.size,6);
      p := p^.link
    END
```

```
END SpeicherListe;

BEGIN                              (* Modul-Initialisierung TeStorage *)
   WriteString("<TeStorage wird installiert...");
   MasterCount := 0;
   StoreList := NIL;
   WriteString("Ende TeStorage>");
END TeStorage.
```

Bei der Entwicklung der komplexen verzeigerten Struktur der Module in Kapitel 5 wurde erfolgreich von `TeStorage` Gebrauch gemacht.

2.3 Die Behandlung von Dateien

Bis jetzt haben wir unsere Daten im Speicher gehalten. Variablen wurden vom Programm vorbelegt oder über die Tastatur eingegeben. Für professionelle Anwendungszwecke benötigen wir Werkzeuge zum Speichern und Abholen von Daten aus Dateien, die sich auf externen Massenspeichern (Diskette, Festplatte, demnächst auch optische Platte) befinden. Im weiteren Sinne zählt man zu diesen Dateien auch periphere Ein- und Ausgabemedien wie Tastatur, Drucker (ein echtes »write-only«-Medium) oder Modem.

Daher ist in diesem Zusammenhang auch der Modul `InOut` zu nennen, mit dem man die Ein-/Ausgabe umleiten kann. Hier gibt es die Prozedur `RedirectOutput`, mit der man den Drucker als Ausgabemedium anwählen kann. `InOut` dient also zu Ein-Ausgabe von Text und Zahlen auf Standarddateien.

In Modula gibt es keine Datenstruktur für Dateien, die zum Sprachumfang selbst gehört. Die Dateiverarbeitungs-Werkzeuge sind in externe Module ausgelagert.

Bereits der Schöpfer der Sprache *N. Wirth* schlägt hier die Standardmodule `Streams` und `Files` vor. `Streams` ist für das byte-weise sequentielle Lesen und Schreiben bei Dateien zuständig, man spricht von einem Strom von Bytes (Datenabstraktion). Der hierauf aufbauende Modul `Files` ermöglicht größere Dateioperationen, wie man sie von anderen Sprachen wie Pascal her kennt. Es werden hier Lese- und Schreiboperationen für Standard-Datentypen bereitgestellt. Leider differieren die Module zur Dateiverarbeitung bei jedem Modula-System für den Atari. Hier eine kurze Übersicht:

TDI-Modula hält sich im großen und ganzen an das von *Wirth* vorgeschlagene Konzept. Es gibt die Module mit dem Namen `Streams` und `TextIO`. Ähnlich bei Hänisch-Modula, dessen Bibliothek sich stark an die Definition von *Martin Odersky* anlehnt. Es werden die Module `Files` und `TextIO` bereitgestellt.

Bei SPC-Modula findet man die Module `ByteStreams` für sequentielle byte- und word-weise I/O-Operationen (I/O steht für *input/output*, »Ein-/Ausgabe«) und `Textstreams`. Letzterer erlaubt die Ein- und Ausgabe auf ASCII-Files ähnlich den Prozeduren aus `InOut`. Weiterhin gibt es `FileSystem` für Dateien mit Direktzugriff *(random file access)*, wobei gleichzeitiges Lesen und Schreiben auf jede Stelle des Files möglich ist. `Textfiles` unterstützt Textdateien mit Direktzugriff. Das Einfügen (*insert*) von Zeichen in das File ist möglich.

Das MSM2-System bietet die Module `Streams`, `Files` und `FileIO`. Letzterer implementiert die Ein-/Ausgabe auf Dateien und ist im großen und ganzen mit `InOut` kompatibel, was recht bequem ist.

Zum Megamax-Dateisystem gehören: `Files` zum Öffnen und Schließen von Dateien. `Binary` für byte-weise (nicht textuelle) Ein-/Ausgabe von Daten auf Files. Dieser Modul entspricht am ehesten `Streams`, jedoch ist Direktzugriff mit der Prozedur `Seek` möglich. `Text` dient für die Ein-/Ausgabe von Textdateien auf beliebige Daten; `NumberIO` für Ein-/Ausgabe von Zahlen in Textfiles auf beliebige Dateien.

Die folgenden Beispielprogramme beziehen sich wieder auf Megamax-Modula. Da sie ausführlich erklärt werden, dürfte es keine Schwierigkeit bereiten, sie für ein anderes System umzuschreiben.

2.3.1 Einführende Beispiele

Das Einfachste, was man mit Diskettendateien machen kann, ist:

1. die Datei mit einem bestimmten Namen öffnen
2. sequentiell etwas Abspeichern
3. die Datei wieder schließen

Ein zweites Programm kann dann diese Datei wieder öffnen, die Daten in den Kernspeicher lesen. Als »Daten« nehmen wir im ersten Beispiel einen Text, im zweiten die Zahlen 1 bis 10.

```
MODULE SchreibeTextAufDatei;

FROM InOut IMPORT Read, ReadString, WriteString, WriteLn;
FROM Files IMPORT Create, Close, File, Access, ReplaceMode;
IMPORT Text;

VAR f            : File;
    FileName, s  : ARRAY  [0..80] OF CHAR;
    taste        : CHAR;
    i            : CARDINAL;
```

```
BEGIN
  WriteString("Demonstration des Schreibens auf eine Textdatei"); WriteLn;
  WriteString("Geben Sie den kompletten Pfadnamen für die Datei an: ");
  ReadString(FileName);
  Create(f,FileName,writeSeqTxt,replaceOld);
  FOR i:=0 TO 3 DO
    WriteString("Text: "); ReadString(s);
    Text.WriteString(f,s);
    Text.WriteLn(f)
  END;
  Close(f);
  WriteString("Ihren Text können Sie mit dem Programm 'LiesText' lesen.");
  Read(taste);
END SchreibeTextAufDatei.
```

Die vier Textzeilen, die man mit diesem Programm geschrieben hat, kann man mit dem Editor ansehen. Oder auch mit dem folgenden Programm:

```
MODULE LiesTextVonDatei;

FROM InOut IMPORT Read, ReadString, WriteString, WriteLn;
FROM Files IMPORT Open, Close, File, Access, EOF;
IMPORT Text;

VAR f           : File;
    FileName, s : ARRAY [0..80] OF CHAR;
    taste       : CHAR;

BEGIN
  WriteString("Demonstration des Lesens von einer Textdatei"); WriteLn;
  WriteString("Geben Sie den kompletten Pfadnamen für die Datei an: ");
  ReadString(FileName);
  Open(f,FileName,readSeqTxt);
  WHILE NOT EOF(f) DO
    Text.ReadString(f,s);
    WriteString(s);
    WriteLn
  END;
  Close(f);
  Read(taste);
END LiesTextVonDatei.
```

Das Programm kann natürlich auch andere Textfiles lesen, zum Beispiel Ihre Modula-Programmtexte.

Mittels der Routinen aus `NumberIO` können auch Zahlen in die Textdatei geschrieben werden:

```
MODULE SchreibeZahlenAufDatei;

FROM InOut IMPORT Read, ReadString, WriteString, WriteCard, WriteLn;
FROM Files IMPORT Create, Close, File, Access, ReplaceMode;

IMPORT NumberIO;
IMPORT Text;

VAR f        : File;
    FileName : ARRAY  [0..80] OF CHAR;
    taste    : CHAR;
    i        : CARDINAL;

BEGIN
  WriteString("Demonstration des Speichern von Zahlen auf eine Datei"); WriteLn;
  WriteString("Geben Sie den kompletten Pfadnamen für die Datei an: ");
  ReadString(FileName);
  Create(f,FileName,writeSeqTxt,replaceOld);
  FOR i:=0 TO 10 DO
    WriteCard(i,3); WriteLn;
    NumberIO.WriteCard(f,i,3);
    Text.WriteLn(f)
  END;
  Close(f);
  WriteString("Diese Zahlen können Sie mit dem Programm 'LiesZahl' lesen.");
  Read(taste);
END SchreibeZahlenAufDatei.
```

Zum Lesen nun:

```
MODULE LiesZahlenVonDatei;

FROM InOut IMPORT Read, ReadString, WriteString, WriteLn, WriteCard;
FROM Files IMPORT Open, Close, File, Access, EOF;
IMPORT NumberIO;

VAR f        : File;
```

```
    FileName : ARRAY [0..80] OF CHAR;
    i        : CARDINAL;
    taste    : CHAR;
    ok       : BOOLEAN;

BEGIN
  WriteString("Demonstration des Lesens von Zahlen aus einer Datei"); WriteLn;
  WriteString("Geben Sie den kompletten Pfadnamen für die Datei an: ");
  ReadString(FileName);
  Open(f,FileName,readSeqTxt);
  WHILE NOT EOF(f) DO
    NumberIO.ReadCard(f,i,ok);
    IF ok THEN WriteCard(i,6) END;
  END;
  Close(f);
  Read(taste);
END LiesZahlenVonDatei.
```

Im nächsten Abschnitt sieht man, wie Verbunde, die verschiedene Datentypen enthalten, abzuspeichern sind. Außerdem wird hier gezeigt, wie schnell auf einen bestimmten Datensatz zugegriffen werden kann, ohne jedesmal die gesamte Datei sequentiell zu durchlaufen.

2.3.2 Die Verwaltung einer Datei mit einem Baum

Wir haben im Vorwort versprochen, sowohl für den Anfänger als auch für den Fortgeschrittenen interessant zu bleiben. Aus diesem Grund geht es nun im Niveau steil hoch zu einer professionellen Anwendung. Dieser Abschnitt wird ein relativ komplexes Programm bringen. Um nun nicht gleich den Anfänger zu entmutigen, sei gesagt, daß es ab dem nächsten Abschnitt wieder mit wesentlich reduziertem Schwierigkeitsgrad weitergeht. Legen Sie also das Buch nicht gleich weg, wenn Sie beim ersten Lesen mit diesem Abschnitt nicht zurecht kommen. Überschlagen Sie ihn einfach, er wird für den weiteren Verlauf nicht weiter gebraucht! Lesen Sie diesen Abschnitt erneut, wenn Sie in den folgenden Kapiteln mehr Programmiererfahrung gesammelt haben.

Der auf Diskette zu speichernde Datentyp lautet (vgl. Abschnitt 2.1):

```
TYPE KundenTyp = RECORD
                     KundenNr      : CARDINAL;
                     Name,Vorname : str20;
                     Wohnort       : str20
                   END;
```

Zur Verwaltung braucht man folgendes:

1. Möglichkeit der Eingabe eines Kunden und anschließendes Abspeichern in der Kundendatei
2. Möglichkeit des Suchen eines Kunden, Schlüssel ist hierbei der Nachname
3. Möglichkeit des »Vor- und Rückwärtsblätterns« in der Datei
4. Möglichkeit des Löschens des Kunden

Zu 1.

Eingegeben werden die Kunden über eine Eingabemaske für die vier Komponenten des Verbundes. Es ist chic, solche Eingabemasken mit der GEM-Oberfläche des Atari anzufertigen; hierzu kommen wir aber erst im 4. Kapitel. Wir bringen hier einen Modul, der etwas altmodisch Eingaben auf dem Text-Bildschirm gestattet. In Millionen Verwaltungen haben Eingabemasken ein ähnliches Design! Die eingegebenen Personen werden in der Reihenfolge abgespeichert, wie sie ankommen. In der Kundendatei ist alles wild durcheinander. Wir sorgen dafür, das trotzdem alles sortiert aussieht!

Zu 2.

Damit wir in der chaotischen Kundendatei schnell etwas finden, brauchen wir eine geordnete »stützende« Struktur neben ihr. Wir wählen dazu einen geordneten binären Baum mit Rückzeiger, dessen Einträge den Schlüssel `KundenName` und der Position des entsprechenden Datensatzes in der Kundendatei besteht. Hierdurch ist eine schnelle Suchmöglichkeit und Zugriff auf einen Kunden gegeben.

Die hier vorgeschlagene Methode, über einen Baum (auch über eine Listen-oder ARRAY-Struktur) auf einen bestimmten Datensatz einer Datei zuzugreifen, heißt **ISAM** (**I**ndex **S**earch (oder **S**equential) **A**ccess **M**ethod).

In der folgenden Reihenfolge werden beispielsweise Kunden eingegeben. Diese bleiben auf der Datei ungeordnet:

Nachname	Datensatznummer
Klein	1
Duck	2
Zorro	3

Der geordnete Kundenbaum hat dann die Einträge:

Nachname	Datensatznummer
Duck	2
Klein	1
Zorro	3

Wie man an diesem Beispiel sieht, läßt sich eine Datei mit wahlfreiem Zugriff auch über Stützfelder verwalten. Bäume sind aber effizienter beim Einfügen und Löschen.

Zu 3.

Das Vor- und Rückblättern ist durch unsere »Baum mit Rückzeiger«-Struktur nun kein Problem mehr.

Zu 4.

Wenn ein Datensatz gelöscht werden soll, wird man ihn zunächst über den Kundennamen suchen und auf dem Bildschirm anzeigen. In den beiden Bäumen sind die entsprechenden Daten ebenfalls zu löschen. In der Kundendatei entstehen durch das Löschen mit der Zeit »Löcher«, die wir wieder für neu eingegebene Kunden benutzen können. Dazu muß man aber wissen, welche Datensätze in der Datei nicht belegt sind. Also benötigt man noch eine Struktur, die die Löcher festhält. Hier bietet sich ein Stapel an. Wir speichern dabei die Verweise auf den nächsten Datensatz in den gelöschten Datenfeldern.

Aus der Beschreibung dürfte klar geworden sein, daß wir eine recht komfortable, aber auch aufwendige Dateiverwaltung implementierten. Nach dem Leitsatz »ein fertiges Programm ist ein veraltetes Programm«, könnten Sie noch versuchen, einen weiteren Stützbaum für die Kundennummern als zweiten Suchschlüssel anzulegen. Die benötigten Strukturen dafür sind bereits vorhanden!

Die obigen Ausführungen mögen als Entschuldigung dafür reichen, warum das Programm ziemlich lang geraten ist.

```
MODULE DateiverwaltungMitStuetzBaum;

FROM SYSTEM   IMPORT ADDRESS;
FROM Tastatur IMPORT lies, SLinks, SRechts, PHoch, PTief,
                     F1, F2, F3, F4, F5, F6, F7, F8, F9, F10;
FROM Eingabe  IMPORT Glocke, LiesZeichen, LiesWort, LiesCard;
FROM InOut    IMPORT GotoXY, Write, WriteString, WriteCard, WritePg, FlushKbd,
                     ReadString;
FROM Strings  IMPORT String, Compare, Relation;

IMPORT Files, Binary;
```

```
IMPORT Baum;

CONST   (* >>>>>>>>>>>   Bitte nach Bedarf ändern             *)
        FileFuerDaten = "F:\TEST.DAT";      (* Pfad für Hauptdatei *)
        FileFuerNamen = "F:\TEST.NAM";      (* Pfad für Stützdatei *)

CONST RET = 13;  ESC = 33C;

TYPE

     Str20     = ARRAY [0..19] OF CHAR;
     FileIndex = LONGINT;
     KundenTyp = RECORD
                   KundenNr       : CARDINAL;
                   Name, Vorname  : Str20;
                   Wohnort        : Str20;
                 END;
     NameTyp = RECORD
                 Name    : Str20;
                 SatzNr  : FileIndex
               END;

VAR Aktu : RECORD
             Kunde: KundenTyp;
             SatzNr: FileIndex;
           END;
    Bank: RECORD
            NamenPath, DatenPath: String;
            NamenFile, DatenFile: Files.File;
            NamenBaum: Baum.TREE
          END;
    wahl  : CARDINAL;

PROCEDURE invers;
BEGIN Write(ESC); Write("p") END invers;

PROCEDURE normal;
BEGIN Write(ESC); Write("q") END normal;

PROCEDURE LoeschZeile(zeile : CARDINAL);
BEGIN GotoXY(0,zeile); Write(ESC); Write("K") END LoeschZeile;

PROCEDURE Meldung(s: ARRAY OF CHAR);
```

```
BEGIN
  LoeschZeile(17); GotoXY(2,17); WriteString(s);
END Meldung;

PROCEDURE Frage(s : ARRAY OF CHAR) : BOOLEAN;
BEGIN
  Meldung(s); WriteString(" (j/n)? ");
  RETURN CAP(LiesZeichen("JjNn")) = "J"
END Frage;

PROCEDURE NamenKleiner(p1,p2: ADDRESS): BOOLEAN;
VAR
    pSt1, pSt2: POINTER TO NameTyp;
BEGIN
    pSt1 := p1; pSt2 := p2;
    RETURN Compare(pSt1^.Name,pSt2^.Name) = less
END NamenKleiner;

(* ======================================================================
   ======================  Datei-Verwaltung ========================
*)
MODULE DateiVerwalter;

IMPORT  FileIndex,KundenTyp, Bank, Baum, NamenKleiner, NameTyp,
        Files, Binary;

EXPORT  LiesKunde, SchrKunde, NeuerKunde, LoescheKunde, KundenAnzahl,
        DatenErzeugen, DatenEinlesen, DatenSichern;

TYPE
    HeaderTyp = RECORD HighIndex,ErsteFreie: FileIndex; Anzahl: CARDINAL END;
    SatzArt = (satzFrei, satzBelegt, satzHeader);
    SatzTyp = RECORD
                CASE Art: SatzArt OF
                  satzFrei  : NaechsteFreie : FileIndex |
                  satzBelegt: Info          : KundenTyp |
                  satzHeader: Header        : HeaderTyp
                END
              END;

VAR
   Header : HeaderTyp;
```

```
PROCEDURE WriteSatz(nr: FileIndex; VAR satz: SatzTyp);
BEGIN
  Binary.Seek(Bank.DatenFile, nr*FileIndex(SIZE(satz)), Binary.fromBegin);
  Binary.WriteBlock(Bank.DatenFile, satz)
END WriteSatz;

PROCEDURE ReadSatz(nr: FileIndex; VAR satz: SatzTyp);
BEGIN
  Binary.Seek(Bank.DatenFile, nr*FileIndex(SIZE(satz)), Binary.fromBegin);
  Binary.ReadBlock(Bank.DatenFile, satz)
END ReadSatz;

PROCEDURE NeuerKunde: FileIndex;
VAR
    satz : SatzTyp;
    frei : FileIndex;
BEGIN
  INC(Header.Anzahl);
  IF Header.ErsteFreie = OD THEN
    INC(Header.HighIndex);
    RETURN Header.HighIndex
  ELSE
    frei := Header.ErsteFreie;
    ReadSatz(frei, satz);
    Header.ErsteFreie := satz.NaechsteFreie;
    RETURN frei
  END
END NeuerKunde;

PROCEDURE LoescheKunde(nr: FileIndex);
VAR satz: SatzTyp;
BEGIN
  satz.Art := satzFrei;
  satz.NaechsteFreie := Header.ErsteFreie;
  Header.ErsteFreie := nr;
  WriteSatz(nr, satz);
  DEC(Header.Anzahl)
END LoescheKunde;

PROCEDURE LiesKunde(nr: FileIndex; VAR kunde: KundenTyp);
VAR satz: SatzTyp;
BEGIN
  ReadSatz(nr, satz);
```

```
  IF satz.Art # satzBelegt THEN HALT END;
  kunde := satz.Info
END LiesKunde;

PROCEDURE SchrKunde(nr: FileIndex; VAR kunde: KundenTyp);
VAR satz: SatzTyp;
BEGIN
  satz.Art  := satzBelegt;
  satz.Info := kunde;
  WriteSatz(nr, satz);
END SchrKunde;

PROCEDURE KundenAnzahl: CARDINAL;
BEGIN RETURN Header.Anzahl END KundenAnzahl;

PROCEDURE LiesNamen;
VAR name: NameTyp;
BEGIN
  Binary.ReadBlock (Bank.NamenFile, name);
  Baum.GebeDaten (Bank.NamenBaum, name)
END LiesNamen;

PROCEDURE SchrNamen;
VAR name: NameTyp;
BEGIN
  Baum.HoleDaten (Bank.NamenBaum, name);
  Binary.WriteBlock (Bank.NamenFile, name)
END SchrNamen;

PROCEDURE DatenErzeugen: BOOLEAN;
BEGIN
  Files.Create(Bank.DatenFile,Bank.DatenPath, Files.readWrite,Files.noReplace);
  IF Files.State(Bank.DatenFile) < 0 THEN RETURN FALSE END;
  Files.Create(Bank.NamenFile,Bank.NamenPath, Files.readWrite,Files.noReplace);
  IF Files.State(Bank.DatenFile) < 0 THEN RETURN FALSE END;
  Baum.Einrichten(Bank.NamenBaum, NamenKleiner);
  Header.HighIndex := 0D;
  Header.ErsteFreie := 0D;
  Header.Anzahl := 0;
  RETURN TRUE
END DatenErzeugen;

PROCEDURE DatenEinlesen: BOOLEAN;
```

```
VAR satz: SatzTyp;
BEGIN
  Files.Open(Bank.DatenFile, Bank.DatenPath, Files.readWrite);
  IF Files.State(Bank.DatenFile) < 0 THEN RETURN FALSE END;
  Files.Open(Bank.NamenFile, Bank.NamenPath, Files.readWrite);
  IF Files.State(Bank.NamenFile) < 0 THEN RETURN FALSE END;
  ReadSatz(OD, satz);                          (* Datenbank-Header einlesen *)
  IF satz.Art # satzHeader THEN HALT END;
  Header := satz.Header;
  Baum.LinearZuBaum(Header.Anzahl, LiesNamen, NamenKleiner, Bank.NamenBaum);
  RETURN TRUE
END DatenEinlesen;

PROCEDURE DatenSichern;
VAR satz: SatzTyp;
BEGIN
  satz.Art := satzHeader;
  satz.Header := Header;
  WriteSatz(OD, satz);                         (* Datenbank- Header sichern *)
  Binary.Seek(Bank.NamenFile, OD, Binary.fromBegin);
  Baum.BaumZuLinear(Bank.NamenBaum, SchrNamen);   (* Baum auf Datei sichern *)
  Files.Close(Bank.DatenFile);
  Files.Close(Bank.NamenFile)
END DatenSichern;

END                          DateiVerwalter;

(* ================================================================*)
PROCEDURE LeerKunde;
BEGIN
  WITH Aktu.Kunde DO
    KundenNr := 9999; Name[0] := OC; Vorname[0] := OC; Wohnort[0] := OC END;
END LeerKunde;

PROCEDURE ZeigeKunde;
BEGIN WITH Aktu.Kunde DO
  LoeschZeile(10);
  LoeschZeile(14);
  GotoXY(2, 10); WriteString("Kundennr.:  "); WriteCard(KundenNr,4);
  GotoXY(40,10); WriteString("Nachname :  "); WriteString(Name);
  GotoXY(2 ,14); WriteString("Vorname  :  "); WriteString(Vorname);
  GotoXY(40,14); WriteString("Wohnort  :  "); WriteString(Wohnort)
END END ZeigeKunde;
```

```
PROCEDURE ZeigeMaske;
VAR name: NameTyp;
BEGIN
  IF Baum.gefunden(Bank.NamenBaum) THEN
    Meldung("");
    ZeigeKunde
  ELSE
    LeerKunde; ZeigeKunde;
    Meldung("<kein Kunde gefunden>")
  END;
  GotoXY(2,7); WriteString("Satz Nr.:"); WriteCard(Aktu.SatzNr, 5);
                WriteString(", von: "); WriteCard(KundenAnzahl(), 5 )
END ZeigeMaske;

PROCEDURE LadeAktuellen;      (* Holt die Datensatz-Nr. aus der Baum und... *)
VAR                          (* ...holt den vollständigen Kunden aus der Datei *)
    name: NameTyp;
BEGIN
  IF Baum.gefunden(Bank.NamenBaum) THEN
    Baum.HoleDaten(Bank.NamenBaum, name);
    Aktu.SatzNr := name.SatzNr;
    LiesKunde(Aktu.SatzNr, Aktu.Kunde);
  ELSE
    Aktu.SatzNr := 0D;
  END
END LadeAktuellen;

PROCEDURE speichern;
VAR name : NameTyp;
    neue : FileIndex;
BEGIN
  neue := NeuerKunde();
  name.Name := Aktu.Kunde.Name;
  name.SatzNr := neue;
  Baum.Einfuegen(Bank.NamenBaum, name);
  SchrKunde(neue, Aktu.Kunde);
  Aktu.SatzNr := neue;
END speichern;

PROCEDURE Einfuegen;
VAR
   nr, EndTaste : CARDINAL;
BEGIN
  LeerKunde; ZeigeKunde;
```

```
  Meldung("Geben Sie einen Kunden ein (<ESC> = fertig)!");
  nr := 0;
  REPEAT
    WITH Aktu.Kunde DO
      CASE nr OF
        0 : LiesCard(14,10,1,9999,KundenNr,EndTaste) |
        1 : LiesWort(52,10,20,Name,EndTaste)         |
        2 : LiesWort(14,14,20,Vorname,EndTaste)      |
        3 : LiesWort(52,14,20,Wohnort,EndTaste)
      END
    END;
    CASE EndTaste OF
      PHoch   : IF nr < 2  THEN Glocke ELSE DEC(nr,2) END |
      PTief   : IF nr > 1  THEN Glocke ELSE INC(nr,2) END |
      SRechts : IF ODD(nr) THEN Glocke ELSE INC(nr)   END |   (* Shift --> *)
      SLinks  : IF ODD(nr) THEN DEC(nr) ELSE Glocke   END |   (* Shift <-- *)
      RET     : nr := (nr+1) MOD 4;
    END;
  UNTIL EndTaste = 27;
  IF Frage("Abspeichern") THEN speichern ELSE LadeAktuellen END;
END Einfuegen;

PROCEDURE Loeschen;
VAR name: NameTyp;
BEGIN
  IF NOT Baum.gefunden(Bank.NamenBaum) THEN Glocke; RETURN END;
  IF Frage("Wirklich löschen") THEN
    Baum.HoleDaten(Bank.NamenBaum, name);
    IF name.SatzNr # Aktu.SatzNr THEN HALT END;
    Baum.Loeschen(Bank.NamenBaum);
    LoescheKunde(Aktu.SatzNr);
  END;
END Loeschen;

PROCEDURE Suchen;
VAR name: NameTyp;
BEGIN
  Meldung("Welchen Namen suchen: ");
  ReadString(name.Name);
  Baum.Suchen(Bank.NamenBaum, name);
  LadeAktuellen
END Suchen;

PROCEDURE Naechster;
```

```
BEGIN
  Baum.Naechster(Bank.NamenBaum); LadeAktuellen
END Naechster;

PROCEDURE Voriger;
BEGIN
  Baum.Voriger(Bank.NamenBaum); LadeAktuellen
END Voriger;

PROCEDURE Ende;
BEGIN
  Meldung("Bitte etwas Geduld, es werden noch Daten abgespeichert!");
  DatenSichern
END Ende;

BEGIN
  Bank.NamenPath := FileFuerNamen;
  Bank.DatenPath := FileFuerDaten;
  IF    DatenEinlesen() THEN Meldung("Alte Daten Werden gelesen...")
  ELSIF DatenErzeugen() THEN Meldung("Eine neue Datei wird erzeugt...")
  ELSE  Meldung("Dateien defekt oder unvollständig!"); lies(wahl); HALT END;
  REPEAT
    ZeigeMaske;
    invers;               (* Menü invers darstellen *)
    GotoXY( 1,20); WriteString(" Eingabe F1 ");
    GotoXY(14,20); WriteString(" Löschen F2 ");
    GotoXY(27,20); WriteString(" Suchen F3 ");
    GotoXY(39,20); WriteString(" Voriger F4 ");
    GotoXY(52,20); WriteString(" Nächster F5 ");
    GotoXY(66,20); WriteString(" Ende F10 ");
    normal;
    FlushKbd;
    lies(wahl);
    LoeschZeile(20);      (* Menü wegmachen *)
      CASE wahl OF
        F1 : Einfuegen | F2 : Loeschen  | F3 : Suchen   |
        F4 : Voriger   | F5 : Naechster | F10: Ende     |
      ELSE Glocke END
  UNTIL wahl = F10;
END DateiverwaltungMitStuetzBaum.
```

2.4 Hashen, schneller als Sortieren

Im großen und ganzen geht es in diesem zweiten Kapitel um das Abspeichern und schnelle Wiederfinden von Daten. Hierzu wurden Verfahren mittels Felder, Bäumen und Dateien aufgezeigt. Dabei haben wir die Datenmenge stets in irgendeiner raffinierten Weise auf eine sortierte Struktur abgebildet. Möglicherweise stehen Sie mit den Zeigern noch auf Kriegsfuß. Das nun vorgestellte Verfahren des »Hashens« läßt sich sowohl auf das Finden von Daten in ungeordneten Feldern als auch auf ungeordnete Dateien anwenden, ist bei nicht zu großen Datenbeständen sehr schnell und – das ist der Knüller – ganz einfach zu programmieren!

Damit das Wesentliche klar zum Vorschein kommt, demonstrieren wir das Hash-Verfahren an einem Feld, für Dateien funktioniert es analog.

Wir präzisieren die Problemstellung:

Gegeben ist ein ungeordnetes Feld vom Typ `KundenTyp` (vgl. 2.1):

```
VAR feld: ARRAY [0..max] OF KundenTyp;
```

Suchschlüssel sei der Nachname vom

```
TYPE str20 = ARRAY[0..19] OF CHAR;
```

Bevor nun ein neuer Kunde in das noch leere Feld abgespeichert wird, »zerhacken« (engl. *hash* »zerhacken«) wir den Schlüssel etwa mit der folgenden Funktion:

```
PROCEDURE Hash(Schluessel: ARRAY OF CHAR): CARDINAL
VAR hashwert,i: CARDINAL;
BEGIN
  Hashwert := 0;
  i := 0;
  WHILE (i <= HIGH(Schluessel)) AND (Schluessel[i] # 0) DO
    INC(hashwert, ORD(Schluessel[i]));
    INC(i);
  END;
  RETURN hashwert MOD (max + 1)
END Hash;
```

Diese Funktion hat nur den Sinn, einem Schlüssel eine beliebige Zahl aus dem Intervall [0..max] zuzuordnen, die möglichst für verschiedene Schlüssel verschieden sein soll. Diese Zahl nennt sich »Hash-Wert«.

Hat nun ein Kundenname den Hashwert h, so wird er an der Position feld[h] abgespeichert. Wir speichern einen Datensatz also einfach bei seinem Hashwert!

```
h := Hash(kunde.Name);
feld[h] := kunde;
```

Das ist alles! Und wenn man einen Kunden mit dem Namen »Meier« sucht:

```
kunde := feld[Hash("Meier")];
```

und schon haben wir ihn! Abspeichern und Suchen also mit einem kurzen Zugriff! Leider hat die Sache einen kleinen Haken: die »Kollisionen«. Es kann nämlich vorkommen, das zwei verschiedene Schlüssel den selben Hashwert haben! Beispielsweise haben "Modula" und "Madulo" mit obiger Hash-Funktion beide den Hash-Wert 610. Mathematisch gesehen ist Hash also nicht injektiv. Das geht auch gar nicht, da es mehr verschiedene Strings als CARDINAL-Zahlen gibt.

Aber man kann schlecht zwei Datensätze in dem selben Feldelement speichern. Eine einfache Möglichkeit ist es, den nächsten freien Platz zu nehmen, also den Index h so lange zu erhöhen, bis ein freier Platz gefunden ist. Diese Methode nennt sich »lineares Sondieren«. Sie führt aber zu Ballungen in der Feldbesetzung. Eine bessere Verteilung der Daten über das ganze Feld liefert das »quadratische Sondieren«. Dabei erhöht man h nicht um eins, sondern setzt es um eine Quadratzahl weiter: bei jeder Kollision macht h also größere Schritte. Läuft man dabei oben heraus (h>max), macht man wieder unten weiter.

Da die Quadratzahlen 1,4,9,16,25... im Abstand von 3,5,7,9 voneinander liegen, braucht man natürlich nicht dauernd i*i zu berechnen (Multiplikationen benötigen relativ viel Zeit):

```
d := 1;  (* Abstand zur nächsten Quadratzahl initialisieren *)
h := Hash(Schluessel);
WHILE <Feld belegt> DO
  INC(h,d); INC(d,2);
  IF h >= max+1 THEN h := h-(max+1)
END;
```

Beim Suchen nach einem Element geht man analog vor: Ist das Element Hash(Schluessel) von einem anderem Datensatz belegt, muß man durch quadratisches Sondieren weitersuchen.

Die Theorie zeigt, daß man schnell freie Plätze findet, wenn max+1 eine Primzahl ist. Damit nicht zu lange im Feld nach freien Plätzen »rotiert« wird, wählt man das Feld auch gerne ca.

10% größer, als man es eigentlich benötigt. Im Programm wählen wir `max=100`, da 101 eine Primzahl ist.

```
MODULE HashDemo;

FROM RandomGen IMPORT Randomize, RandomCard;
FROM InOut     IMPORT WriteString, WriteLn, ReadString, WriteCard;

IMPORT Strings;

CONST prim  = 101;                          (* Primzahl, Anzahl der Feldelemente *)
      max   = prim-1;

TYPE
    Str4    =  ARRAY [0..3] OF CHAR;
    Verbund =  RECORD
                  Schluessel : Str4;
                  (* Info    : InfoTyp *)              (* hier beliebige Daten *)
               END;

VAR
    feld    : ARRAY [0..max] OF Verbund;
    Element : Verbund;

PROCEDURE DummyFeld;
VAR i : CARDINAL;
BEGIN
  FOR i:=0 TO max DO feld[i].Schluessel := "" END;    (* Kennung für unbelegt *)
END DummyFeld;

PROCEDURE Hash(Schluessel : ARRAY OF CHAR) : CARDINAL;
VAR i, summe : CARDINAL;
BEGIN
  summe := 0;
  i := 0;
  WHILE (i <= HIGH(Schluessel)) AND (Schluessel[i] # 0C) DO
    INC(summe, ORD(Schluessel[i]));
    INC(i)
  END;
  RETURN summe MOD prim
END Hash;

PROCEDURE Einfuegen(Element : Verbund);
VAR h,d : CARDINAL;
```

```
BEGIN
  d := 1; h := Hash(Element.Schluessel);
  WHILE NOT (feld[h].Schluessel[0] = 0C) DO                 (* Platz besetzt *)
    INC(h,d); INC(d,2);                                   (* quadr. Sondieren *)
    IF h >= prim THEN h := h - prim END;
    IF d = prim THEN HALT END;                              (* Feldüberlauf *)
  END;
  feld[h] := Element
END Einfuegen;

PROCEDURE Suchen(VAR Element : Verbund) : BOOLEAN;
VAR h,d : CARDINAL;
BEGIN
  d := 1; h := Hash(Element.Schluessel);
  LOOP
    IF Strings.StrEqual(feld[h].Schluessel,Element.Schluessel) THEN
      Element := feld[h]; RETURN TRUE; EXIT END;               (* gefunden *)
    INC(h,d); INC(d,2);                                   (* quadr. Sondieren *)
    IF h >= prim THEN h := h - prim END;
    IF d = prim THEN RETURN FALSE; EXIT END;
  END;
END Suchen;

PROCEDURE FeldZufaelligBelegen;
VAR i,j : CARDINAL;
BEGIN
  Randomize(0L);
  FOR i := 0 TO max - (max DIV 10) DO      (* ca 10% des Feldes bleibt frei *)
                                         (* Feld mit Zufallsstrings belegen *)
    FOR j := 0 TO 3 DO Element.Schluessel[j] := CHR(RandomCard(65,90)) END;
    Einfuegen(Element);
  END;
END FeldZufaelligBelegen;

PROCEDURE FeldDrucken;
VAR i : [0..max];
BEGIN
  FOR i := 0 TO max DO WriteCard(i,3);
    IF feld[i].Schluessel[0] = 0C THEN WriteString("----");
      ELSE WriteString(feld[i].Schluessel); WriteString(" ")
    END
  END
END FeldDrucken;
```

```
BEGIN
  DummyFeld;
  FeldZufaelligBelegen;
  FeldDrucken;
  LOOP
    WriteLn; WriteString("Nach welchem Schluessel suchen (RET = Ende) ");
    ReadString(Element.Schluessel);
    IF Element.Schluessel[0] = 0C THEN EXIT END;
    IF Suchen(Element) THEN
      WriteString("gefunden --> "); WriteString(Element.Schluessel)
    ELSE
      WriteString("Es ist kein Element mit diesem Schlüssel gespeichert!")
    END;
  END;
END HashDemo.
```

Das Programm demonstriert das Einfügen von Schlüsseln – hier Zufallszeichenketten – sowie das anschließende Suchen.

Das Löschen eines Elementes `feld[i]` geschieht einfach, indem man es durch Zuweisen des leeren Strings "" als »leer« markiert (`feld[i].Schluessel := ""`).

Statt unserer Hash-Funktion kann man auch eine beliebig andere Hash-Funktion benutzen; Hauptsache, sie liefert bei dem selben Schlüssel immer den selben und sonst möglichst einen anderen Wert.

Insgesamt sehen wir, daß Hashen ein sehr einfaches und effizientes Verfahren ist. Es kommt in vielen Fällen der Praxis zu Anwendung.

Noch etwas Kritisches: Wenn man nur Daten abspeichern und sie wiederfinden will – was einen häufigen Anwendungsfall darstellt –, ist Hashen ideal. Problematisch wird es aber, wenn das gesamte Feld sortiert ausgegeben werden soll.

Abschließend noch eine andere Möglichkeit der Kollisionsauflösung. `feld` sei ein Feld von Stapeln. Soll ein Datensatz mit dem Hash-Wert `h` eingefügt werden, fügt man ihn einfach in den Stapel `feld[h]` ein.

Kollisionen stören so nicht mehr, da auf jeden Stapel beliebig viele Elemente abgelegt werden können. Aber wir wollten ja hier einmal etwas ohne Zeiger schreiben ...

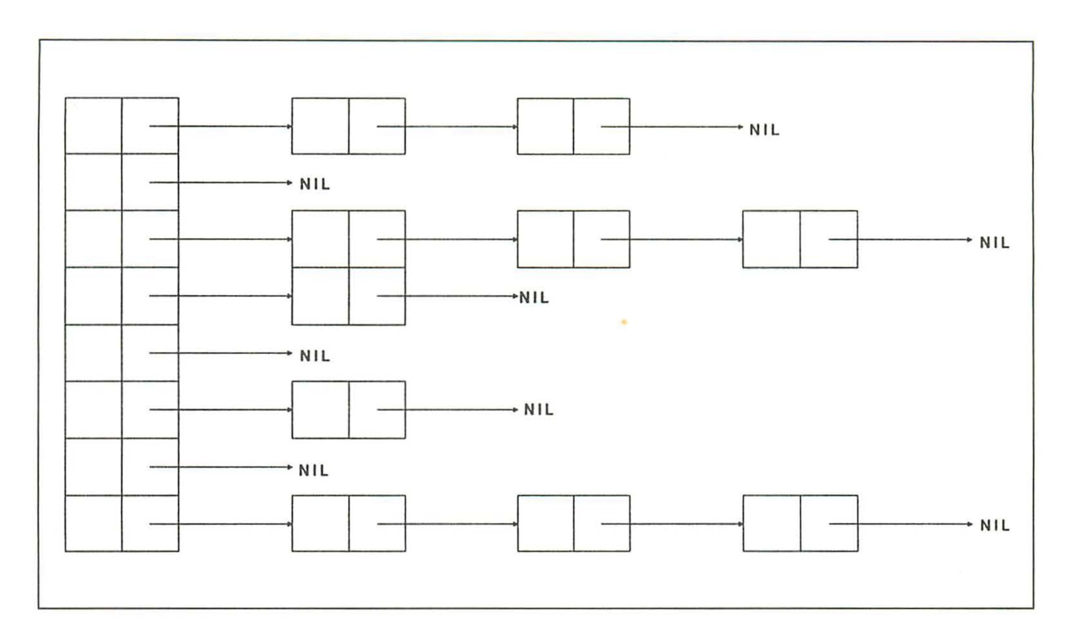


Bild 2.11: Feld von Stapeln

KAPITEL 3

Der 68000-Assembler unter Modula-2

»Pfui«, werden sich einige abwenden, »Assemblerroutinen mitten in Modula-Programmen? Warum soll man sich in die Niederungen der Maschine herablassen, wo Modula doch so flexibel ist?« Stimmt eigentlich, sehen wir uns dazu das folgende Beispiel an:

```
PROCEDURE AusTausch(VAR var1, var2: ARRAY OF BYTE);
VAR hilf : BYTE;
    i    : CARDINAL;

BEGIN
  FOR i := 0 TO HIGH(var1) DO
    hilf := var1[i]; var1[i] := var2[i]; var2[i] := hilf
  END
END AusTausch;
```

Diese Prozedur tauscht zwei Variablen `var1` und `var2` aus und wird zum Beispiel beim Quicksort-Algorithmus gebraucht. Dabei können `var1` und `var2` von beliebigem Datentyp sein. Schön einfach und flexibel, vom Modula-Standpunkt aus.

Doch machen wir uns einmal klar, was der Prozessor beim Abarbeiten des Schleifenrumpfes leisten muß. Dazu muß man wissen, wie der Motorola-68000-Prozessor, das Herz unseres Ataris, im Prinzip arbeitet.

Der Motorola 68000

Dieser Prozessor hat 16 interne Speicher für 32 Bit, sogenannte Register, und zwar acht »Datenregister« D0 bis D7 zum Speichern von Daten und acht Adreßregister A0 bis A7 zur Aufnahme von Adressen. Die Datenregister können mit 8 Bit, 16 Bit oder 32 Bit geladen werden; das paßt genau für die Modula-Datentypen `CHAR`, `CARDINAL` und `LONGCARD`. Die Adreßregister sind 32 Bit (4 Byte) breit. Intern werden davon nur 24 Bit benutzt. Damit ergeben sich 16 Mbyte mögliche Adressen, das dürfte fürs erste reichen...

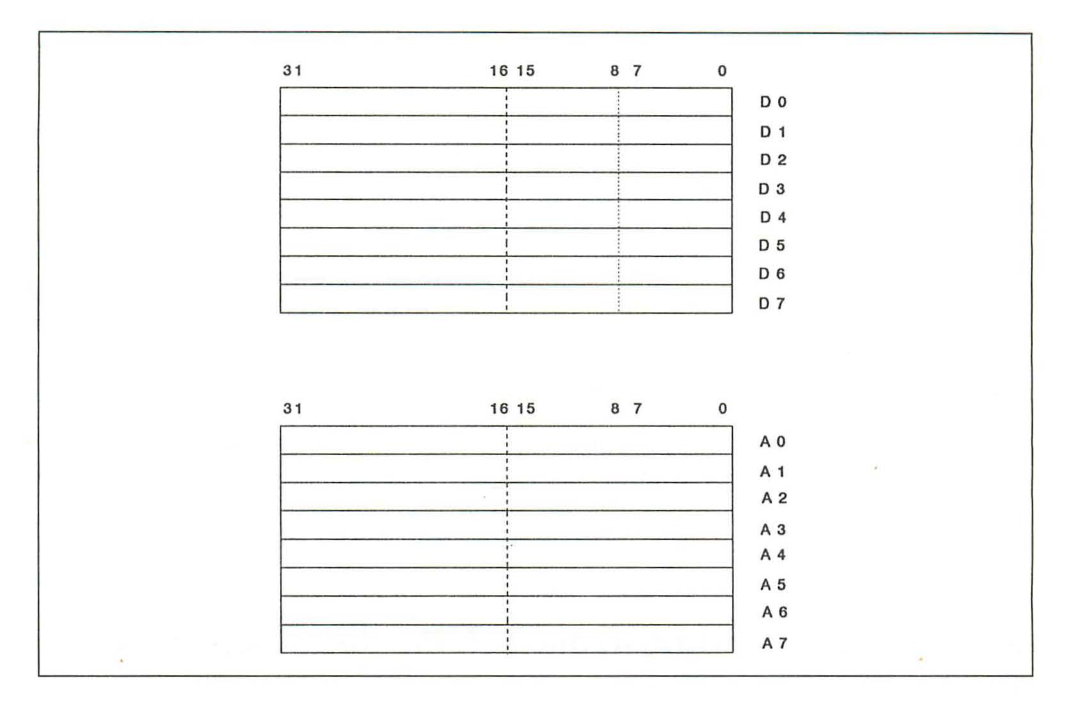


Bild 3.1: Daten- und Adreßregister des MC 68000

Die Abarbeitung der obigen Zuweisungskette

```
hilf := varl[i];
varl[i] := varl[i];
varl[i] := hilf
```

geschieht – wenn der Compiler nicht besonders intelligent ist – nun folgendermaßen:

`hilf:= varl[i]:`

1. Die Adresse von `varl[i]`wird berechnet.
2. Der Wert `varl[i]`wird in ein Datenregister geladen.
3. Der Inhalt des Datenregisters wird nach `hilf` geladen.

`varl[i]:= var2[i]:`

4. Die Adresse von `var2[i]` wird berechnet.
5. Der Wert `var2[i]`wird in ein Datenregister geladen.
6. Die Adresse von `varl[i]`wird erneut berechnet.
7. Der Inhalt des Datenregisters wird an diese Adresse geladen.

`var2[i]:= hilf:`

8. Der Wert von `hilf` wird in ein Datenregister geladen.
9. Die Adresse von `var2[i]`wird erneut berechnet.
10. Der Inhalt des Datenregisters wird an diese Adresse geladen.

Gegebenenfalls wird bei jedem Anweisungsblock noch geprüft, ob der Index `i` für `var2[i]` bzw. `var1[i]` zulässig ist (nennt sich »ABC« *array bound check*= »Feldindexgrenzenüberprüfung«).

So betrachtet, klingt es doch recht kompliziert, insbesondere ist der Umweg über die Variable `hilf` recht unflott. Effizienter wäre es, wenn `var1[i]` und `var2[i]` in zwei verschiedene Datenregister geladen und dann vertauscht zurückgespeichert werden.

Wenn man den Austauschalgorithmus mit dem Ziel angeht, ihn so zu lösen, daß er optimal schnell arbeitet, muß man dieses Programmstück direkt in Assembler programmieren. Hierzu geben wir zunächst einen Überblick über den Befehlsvorrat des Motorola 68000-Prozessors, dem Herzstück des Atari-ST. Es geht also jetzt ans Eingemachte!

3.1 Kurzeinführung in die Befehle des Motorola-68000

Insgesamt verfügt dieser Prozessor über 56 verschiedene Operationen, die man folgendermaßen einteilen kann:

1. Datentransportbefehle (z. B. `MOVE`)
2. Arithmetische Befehle (z. B. `SUB`)
3. Bitmanipulationsbefehle (z. B. `BSET`)
4. Vergleichs- und Testbefehle (z. B. `CMP`)
5. Schiebebefehle (z. B. `LSR`)
6. Logische Befehle (z. B. `AND`)
7. Programmsteuerbefehle (z. B. `BRA`)
8. Systemaufrufe (z. B. `TRAP`)

Wir benötigen für unsere Austausch-Prozedur sowie einige andere kleine, brauchbare Prozeduren von den 56 Befehlen nur die im folgenden aufgeführten. Hierbei steht ».s« für ».B« `(BYTE)`, ».W«`(WORD)` oder ».L«`(LONGWORD)`, je nachdem, ob der Operand 1 Byte, 2 Byte oder 4 Byte umfaßt. Fehlt diese Angabe, wird ».W« angenommen: `MOVE D0,D1` entspricht `MOVE.W D0,D1` und kopiert ein `WORD (16Bit)`.

Aus der 1. Gruppe (Datentransportbefehle):

MOVE.s *<quelle>*, *<ziel>* *move*, »transportieren«
Hier wird *<quelle>* nach *<ziel>* kopiert.
In Modula etwa: *<ziel>*:= *<quelle>*

CLR.s Dn *clear*, »löschen«
setzt Datenregister n auf 0.
In Modula etwa: Dn := 0

Aus der 2. Gruppe (arithmetische Befehle):

ADDQ.s#k, *<ziel>* *add quick*, »schnelle Addition«
Die Konstante k wird zu *<ziel>* zuaddiert; k = 1..8
In Modula etwa: INC(Dn,k)

SUBQ.s#k, *<ziel>* *subtract quick*, »schnelle Subtraktion«
Die Konstante k wird von *<ziel>* abgezogen; k = 1..8
In Modula etwa: DEC(Dn,d)

Aus der 5. Gruppe (Schiebebefehle):

LSL.s Dn,Dm *logical shift left*, »Verschiebung nach rechts«
Die Bits im Datenregister Dm werden um so viele Stellen nach links verschoben, wie Dn angibt.
Etwa vergleichbar mit: $Dm := Dm * 2^{Dn}$

LSR.s Dn,Dm *logical shift right*, »Verschiebung nach links«
Die Bits im Datenregister Dm werden um so viele Stelle nach rechts verschoben, wie Dn angibt.
Etwa vergleichbar mit: $Dm := Dm\ DIV 2^{Dn}$

Aus der 6. Gruppe (logische Befehle):

AND.s Dn,Dm *and*, logisches »UND«
Das Datenregister Dm wird mit dem Register Dn bitweise mit »UND« verknüpft. Das Ergebnis steht anschließend in Dm.

OR.s Dn,Dm *or*, logisches »ODER«
Das Datenregister Dm wird mit dem Register Dn bitweise mit »ODER« verknüpft. Das Ergebnis steht anschließend in Dm.

EOR.s Dn,Dm *exclusive or*, »entweder ODER«
Das Datenregister Dm wird mit dem Register Dn bitweise mit »exclusive-ODER« verknüpft. Das Ergebnis steht anschließend in Dm.

`NOT.s Dn` *not*, »nicht« (Komplement)
Die Einsen in diesem Datenregister werden zu Nullen und umgekehrt.

Aus der 7. Gruppe (Programmsteuerbefehle):

`BRA <Label>` *branch always*, »verzweigen«
Normalerweise wird ein Assemblerprogramm in der Reihenfolge seiner Befehle abgearbeitet. Mit `BRA` ist der »Sprung« zu einer anderen Programmstelle möglich, die mit einer Sprungmarke *<Label>* gekennzeichnet ist.

`DBF Dn, <Label>` *decrement and branch*, »Dekrement und Sprung«
Erniedrigt `Dn` (als `WORD!`) um eins und springt dann nach *<Label>*, falls `Dn` nicht vorher Null war. Eignet sich zur Implementation von schnellen Schleifen.

Aus der 8. Gruppe (Systemaufrufe):

`TRAP #n` *trap*, »Falle«; Falltür zum Betriebssystem
Der Prozessor startet eine Ausnahmeprozedur. Bei dem Atari sind:

`TRAP #1:`	GEMDOS-Aufruf
`TRAP #2:`	GEM-Aufruf (VDI,AES)
`TRAP #13:`	BIOS-Aufruf
`TRAP #14:`	XBIOS-Aufruf

Die Besprechung sämtlicher 56 Befehle sprengt den Rahmen dieses Buches, zumal die einzelnen Operationen mit bis zu 14 (!) verschiedenen Adressierungsarten vorgenommen werden können. Damit sind die Argumente der Befehle gemeint. Unsere Liste beschränkt sich auf diejenigen Adressierungen, die wir im folgenden benötigen. Wer sich hier weiterarbeiten will, dem sei das Buch [V] empfohlen; zur Orientierung und zum Nachschlagen verweisen wir auf Anhang C.

3.2 Assembler-Anweisungen in Modula-2-Routinen

Wie kann man nun in Assembler geschriebene Routinen in seine Modula-Texte einbinden? Bei TDI- und SPC-Modula geht man wie folgt vor:

Man tippt die Routinen in ein gesondertes Assemblerprogramm und testet sie dort aus. Nach dem Assemblieren erhält man reinen Maschinencode, der aus 16-Bit-Instruktionen besteht, vom Modula Standpunkt aus sind das CARDINAL-Zahlen. Nun schreibt man ein Modula-Programm und setzt diese Zahlen in eine beliebig lange Parameterliste der Prozedur `CODE` (bei TDI) oder `INLINE` (bei SPC-Modula), die jeweils aus `SYSTEM` importiert werden. In Hänisch-Modula wird für jedes Assemblerwort eine `CODE`- oder `LOAD`-Anweisung aufgerufen werden (`CODE` und `LOAD` sind Prozeduren aus `SYSTEM`).

Deutlich eleganter geht das nun bei Megamax-Modula und MSM2. Dort kann man direkt Assembleranweisungen, also die oben angegebene Mnemonic-Form der Befehle, in den Modula-Text einfügen. Diese Stellen werden in Megamax-Modula von `ASSEMBLER` und `END`, bei MSM2 von `(*$A+*)` und `(*$A-*)` geklammert:

Bei MSM2:

```
(*$A+*)
   <Assembler-Anweisung>
   <Assembler-Anweisung>
   <...>
(*$ A-*);
```

Bei Megamax-Modula:

```
ASSEMBLER
   <Assembler-Anweisung>
   <Assembler-Anweisung>
   <...>
END;
```

Assembler-Anweisungen gehören bei Megamax-Modula und MSM2 zum Sprachumfang. Nun muß man nur noch wissen, wie der Assemblerteil mit dem übrigem Modula kommuniziert, insbesondere wie auf Variablen und Parameter von Prozeduren zugegriffen wird. Variablen kann man direkt ansprechen:

```
VAR Faktor,Produkt: CARDINAL;

BEGIN    (* Assembler in MSM2 *)
(*$ A+*)
    MOVE.W    Faktor,DO
    LSL       #3,DO
    MOVE.W    DO,Produkt
(*$ A-*);
```

```
VAR Faktor,Produkt: CARDINAL;
BEGIN    (* Assembler in Megamax-Modula *)
ASSEMBLER
    MOVE.W    Faktor,DO
    LSL       #3,DO
    MOVE.W    DO,Produkt
END;
```

Nun, eigentlich ist es wirklich nicht nötig, in Modula auf Assembler zurückzugreifen. Aber wenn man nicht vor hat, seine Programme noch auf anderen Systemen zu verwenden und wenn es wirklich einmal um eine zeitkritische Anwendung geht, sollte der Zugang zum Assembler nicht verwehrt bleiben. Guter Stil ist es aber, die Assemblerteile in eigene Prozeduren zu packen und diese in ein gesondertes Modul zu stecken.

Damit nun für einen Prozeduraufruf nicht mehr zuviel Zeit für den nun nicht mehr nötigen Parameter-Übernahmemechanismus vergeudet wird, läßt sich dieser in Megamax-Modula mit der Option `(*$ L-*)` abschalten. Die Parameter muß man nun »von Hand« abholen. Sie befinden sich auf einem gesonderten Stack, der über das Register A3 verwaltet wird. Wegen dieser Besonderheiten folgen die weiteren Beispiele in Megamax-Modula:

```
PROCEDURE p(n1,n2: CARDINAL, n3:LONGCARD);
(*$ L-*)
BEGIN
  ASSEMBLER
        MOVE.L  -(A3),D0     ; n3 (4 Byte) jetzt in D0
        MOVE.W  -(A3),D1     ; n2 (2 Byte) jetzt in D1
        MOVE.W  -(A3),D2     ; n1 (2 Byte) jetzt in D2
        <...>
  END
END p;
(*$ L+*)
```

Man erkennt hier folgendes:

1. Die Abschaltung der automatischen Parameter-Übernahme `(*$ L-*)` wird mit `(*$ L+*)` wieder eingeschaltet.
2. Hinter Assembleranweisungen können in der selben Zeile Kommentare folgen. Sie werden durch ein Semikolon eingeleitet und reichen bis zum Zeilenende.
3. Mit dem Befehl
   `MOVE.L  -(A3),D0`
   wird ein Parameter von dem Parameter-Stack abgeholt und nach D0 geladen. Die Klammerung der Quelle »`-(A3)`« deutet an, daß der Operand nicht der Wert des Registers `A3` selbst ist; vielmehr enthält A3 die Adresse des Operanden (indirekte Adressierung). Das Minuszeichen besagt, daß die Adresse vor der Verarbeitung noch zu erniedrigen ist und zwar um 4 bei einem `LONGWORD(L)`, 2 bei einem `WORD(W)`und 1 bei einem `BYTE (B)`. Man spricht von »Indirekter Adressierung mit Predekrement«.

4. Die Parameter liegen in umgekehrter Reihenfolge auf dem Stack, also das letzte Argument der Parameterliste muß zuerst abgeholt werden.

5. Bei der `(*$ L-*)`-Option müssen alle Parameter abgeholt werden, die in der Parameterliste stehen, selbst wenn sie nicht benötigt werden! Sonst stimmt der Parameterstack `(A3)` nicht mehr.

Bei VAR-Parametern wird die Adresse (4 Byte) übergeben, unabhängig von der Größe des Parameter:

```
PROCEDURE p(VAR zeichen: CHAR);
(*$ L-*)
BEGIN
  ASSEMBLER
        MOVE.L  -(A3),A0     ; Adresse (4 Byte) von zeichen in A0
        MOVE.B  (A0),D0      ; 'zeichen' jetzt in D0
        <...>
        MOVE.B  D0,(A0)      ; Verändert den VAR-Parameter 'zeichen'
  END
END p;
(*$ L+*)
```

Bei einem offenem Feld als Werte- oder VAR-Parameter wird immer die Adresse und anschließend der HIGH-Wert übergeben; man muß also quasi zwei Parameter abholen:

```
PROCEDURE p(feld: ARRAY OF BYTE);
(*$ L-*)
BEGIN
   ASSEMBLER
         MOVE.W  -(A3),D0     ; HIGH(feld) nach D0
         MOVE.L  -(A3),A0     ; Adresse von 'feld' nach A0
         <...>
   END
 END p;
 (*$ L+*)
```

3.2.1 Modul »LowLevel« für speicherbezogene Operationen

Der erste Modul stellt Prozeduren für das schnelle Kopieren, Austauschen und Füllen von Variablen, sprich Speicherbereichen bereit. Es wird also an das Eingangsbeispiel zu diesem Kapitel angeknüpft. Die Prozedur `CopyN` wird bereits in Kapitel 2.1 benutzt. Zunächst der Definitionsmodul:

```
DEFINITION MODULE LowLevel;

FROM SYSTEM IMPORT ADDRESS, BYTE;

PROCEDURE CopyVar(VAR quelle,ziel: ARRAY OF BYTE);
 (*
  * Kopiert 'quelle' nach 'ziel'; diese Variabeln
  * müssen vom selben Variablentyp sein.
  *)

PROCEDURE CopyN(AdrQuelle,AdrZiel: ADDRESS; anzahl: LONGCARD);
 (*
  * Kopiert einen Speicherbereich der Größe 'anzahl' Bytes. Der
  * Quellbereich beginnt bei 'AdrQuelle', der Zielbereich bei 'AdrZiel'.
  *)

PROCEDURE SwapN(ptrA,ptrB: ADDRESS; anzahl: LONGCARD);
 (*
  * Vertauscht die Speicherbereiche der Größe 'anzahl' Bytes,
  * auf die ptrA und ptrB zeigen.
  *)

PROCEDURE FillBytes(VAR Feld: ARRAY OF BYTE; FuellByte: BYTE);
 (*
  * Fuellt das gesamte 'Feld' mit 'FuellByte'.
  *)

END LowLevel.
```

Der Implementationsmodul ist ausreichend kommentiert:

```
IMPLEMENTATION MODULE LowLevel;

FROM SYSTEM IMPORT ADDRESS, BYTE;

(*$ L-*)
```

```
PROCEDURE CopyVar(VAR quelle,ziel: ARRAY OF BYTE);
BEGIN
  ASSEMBLER
        MOVE.W  -(A3),D1        ; HIGH(ziel)
        MOVE.L  -(A3),A1        ; Adresse von ziel
        MOVE.W  -(A3),D0        ; HIGH(quelle)
        MOVE.L  -(A3),A0        ; Adresse von quelle
    Schleife:
        MOVE.B  (A0)+,(A1)+     ; Byte kopieren, dann A0,A1 erhöhen
        DBF     D1,Schleife     ; zu 'Schleife', wenn nicht fertig
  END
END CopyVar;

PROCEDURE CopyN(AdrQuelle,AdrZiel: ADDRESS; anzahl: LONGCARD);
BEGIN
  ASSEMBLER
        MOVE.L  -(A3),D1        ; anzahl -> D1
        MOVE.L  -(A3),A1        ; AdrZiel -> A1
        MOVE.L  -(A3),A0        ; AdrQuelle -> A0
        BRA     ErsteMal
    Schleife:
        MOVE.B  (A0)+,(A1)+     ; Ein Byte Quelle -> Ziel
    ErsteMal:
        DBF     D1,Schleife     ; Schleife fertig?
  END
END CopyN;

PROCEDURE SwapN(ptrA,ptrB: ADDRESS; anzahl: LONGCARD);
BEGIN
  ASSEMBLER
        MOVE.L  -(A3),D1        ; anzahl -> D1 (Schleifenzähler)
        MOVE.L  -(A3),A1        ; Adresse ptrB -> A1
        MOVE.L  -(A3),A0        ; Adresse ptrA -> A0
        BRA     ErsteMal
    Schleife:
        MOVE.B  (A0),D0         ; Byte von ptrA -> D0 (Hilfsspeicher)
        MOVE.B  (A1),(A0)+      ; Byte von ptrB -> ptrA (und ptrA erhöhen)
        MOVE.B  D0,(A1)+        ; Byte von D0 -> ptrB (und ptrB erhöhen)
    ErsteMal:
        DBF     D1,Schleife     ; Schleife fertig?
  END
END SwapN;
```

```
PROCEDURE FillBytes(VAR Feld: ARRAY OF BYTE; FuellByte: BYTE);
BEGIN
  ASSEMBLER
        SUBQ.L  #1,A3          ; A3 korrigieren, da nächster Parameter BYTE
        MOVE.B  -(A3),D0       ; FuellByte -> D0
        MOVE.W  -(A3),D1       ; HIGH(Feld) -> D1
        MOVE.L  -(A3),A0       ; Addr(Feld) -> A0
    Schleife:
        MOVE.B  D0,(A0)+       ; ein Byte von D0 -> Feld, A0 erhoehen
        DBF     D1,Schleife    ; mit nächstem Byte füllen, wenn nicht fertig
  END
END FillBytes;

(*$ L+*)
END LowLevel.
```

Das folgende kleine Demonstrationsprogramm zeigt eine Anwendung des `LowLevel`-Moduls.

```
MODULE LowLevelTest;

FROM InOut    IMPORT Read, WriteString, WriteLn;
FROM LowLevel IMPORT FillBytes, CopyVar;

VAR i     : CARDINAL;
    s1,s2 : ARRAY[0..19] OF CHAR;
    taste : CHAR;

BEGIN
  FillBytes(s1,"X");
  WriteString("String 's1' mit 20 'X' füllen: ");
  FillBytes(s1,"X"); WriteString(s1); WriteLn;
  WriteString("String 's2' mit 10 'U' füllen: ");
  FillBytes(s2,"U"); s2[10] := 0C; WriteString(s2); WriteLn;
  WriteString("Nun s1 nach s2 kopieren! "); WriteLn;
  CopyVar(s1,s2);
  WriteString("s1: "); WriteString(s1); WriteString(" s2: "); WriteString(s2);
  Read(taste)
END LowLevelTest.
```

3.2.2 Ein Modul für Bitmanipulationen

Vielleicht kennen Sie von Turbo Pascal die logischen Verknüpfungen `AND`, `OR` und `XOR` zwischen ganzen Zahlen. Sie arbeiten bitweise, so ergibt zum Beispiel `13 AND 7` das Ergebnis `5`, denn

$$13_{dezimal} = 1101_{dual}$$
$$7_{dezimal} = 111_{dual}$$
$$13 \text{ AND } 7 = 5_{dezimal} = 101_{dual}$$

Es handelt sich also um bitweise Manipulationen, die man gelegentlich brauchen kann. In diesen Kontext gehört auch `NOT` (Bildung des Einer-Komplements = Invertierung der Nullen und Einsen) und die Prozeduren `SHR` und `SHL` (Verschieben des Bitmusters nach rechts bzw links, was bei Zahlen einer Division durch 2 bzw. eine Multiplikation mit 2 bedeutet.

Ist `i` eine CARDINAL-Variable und $0 \leq n \leq 15$, so kann man

`SHL(i, n)`	statt	`i*m`	$(m = 2^n)$
`SHR(i, n)`	statt	`i DIV m`	$(m = 2^n)$
`AND(i,n-1)`	statt	`i MOD m`	$(m = 2^n)$

benutzen. Wir fassen diese Prozeduren als ein weiteres Beispiel zur Nutzung der Maschinensprache in den Modul `Bitmanipulation` zusammen:

```
DEFINITION MODULE Bitmanipulation;

FROM SYSTEM IMPORT WORD;

PROCEDURE shl(VAR c: WORD; schub: CARDINAL);
(* schiebt c um schub Bits nach links *)

PROCEDURE shr(VAR c: WORD; schub: CARDINAL);
(* schiebt c um schub Bits nach rechts *)

PROCEDURE and(a,b: WORD) : WORD;
(* bitweises AND *)

PROCEDURE or(a,b:WORD): WORD;
(* bitweises OR *)

PROCEDURE xor(a,b: WORD) : WORD;
(* bitweises XOR *)
```

```
PROCEDURE not(a:WORD): WORD;
(* Einerkomplement *)

END Bitmanipulation.
```

```
IMPLEMENTATION MODULE Bitmanipulation;

FROM SYSTEM IMPORT WORD;

(*$ L-*)
PROCEDURE shl(VAR c: WORD; schub: CARDINAL);
BEGIN
  ASSEMBLER
        MOVE    -(A3),D1        ; D1: schub
        MOVE.L  -(A3),A0        ; A0: ADR(c)
        MOVE    (A0),D0         ; D0: c
        LSL     D1,D0           ; D0 := D0 shr D1
        MOVE    D0,(A0)         ; c zurückschreiben
  END
END shl;

PROCEDURE shr(VAR c: WORD; schub: CARDINAL);
BEGIN
  ASSEMBLER
        MOVE    -(A3),D1        ; D1: schub
        MOVE.L  -(A3),A0        ; A0: ADR(c)
        MOVE    (A0),D0         ; D0: c
        LSR     D1,D0           ; D0 := D0 shr D1
        MOVE    D0,(A0)         ; c zurückschreiben
  END
END shr;

PROCEDURE and(a,b: WORD) : WORD;
BEGIN
  ASSEMBLER
        MOVE    -(A3),D1        ; D1: b
        MOVE    -(A3),D0        ; D0: a
        AND     D1,D0           ; D0 := D0 AND D1
        MOVE    D0,(A3)+        ; Funktionswert zurückgeben
  END
END and;
```

```
PROCEDURE or(a,b:WORD): WORD;
BEGIN
  ASSEMBLER
        MOVE    -(A3),D1          ; D1: b
        MOVE    -(A3),D0          ; D0: a
        OR      D1,D0             ; D0 := D0 OR D1
        MOVE    D0,(A3)+          ; Funktionswert zurückgeben
  END
END or;

PROCEDURE xor(a,b: WORD) : WORD;
BEGIN
  ASSEMBLER
        MOVE    -(A3),D1          ; D1: b
        MOVE    -(A3),D0          ; D0: a
        EOR     D1,D0             ; XOR heisst beim 68000 EOR!
        MOVE    D0,(A3)+          ; Funktionswert zurückgeben
  END
END xor;

PROCEDURE not(a:WORD): WORD;
BEGIN
  ASSEMBLER
        NOT   -2(A3)              ; Invertiert a auf dem Stack
  END
END not;

(*$ L+*)
END Bitmanipulation.
```

Es sei noch erwähnt, daß man die Prozeduren `and`, `or`, `xor` und `not` in Modula unter Verwendung von `BITSET` schreiben kann. Das gilt nicht für `shr` und `shl`. SPC-Modula hat im Pseudomodul `SYSTEM` eine Prozedur `SHIFT`, die sowohl `shr` als auch `shl` abdeckt und auf allen Typen arbeitet. Zum Abschluß dieses Einblicks in die Assemblerprogrammierung bringen wir noch ein kleines Beispiel, was sich mit Kryptographie (= Verschlüsselungsverfahren) befaßt.

Nehmen wir an, Sie wollen ein Adventure-Spiel schreiben. Sicherlich werden hier Strings vorkommen, die die Auflösung ihres sorgfältig programmierten Spiels verraten könnten, wenn sich ein cleverer Hacker ihr Programm mit einem Disk-Monitor ansieht. Das gleiche gilt für den Schutz von jeglichen Copyright-Strings oder Firmenbezeichnungen usw. im Code-File. Kodieren Sie diese verräterischen Texte doch einfach, laden den Code in Ihr Programm ein und dekodieren ihn dort wieder.

Bei der vorgestellten Lösung wird der Code als ARRAY OF BYTE übergeben. Der Witz ist nun, daß ein und dieselbe Prozedur das Kodieren und Dekodieren übernimmt. Hier nun das Demonstrationsprogramm:

```
MODULE VerschluesselnEntschluesseln;(* Zeigt eine Anw. der Bitmanipulationen *)

FROM SYSTEM          IMPORT BYTE;
FROM Bitmanipulation IMPORT xor, and;
FROM InOut           IMPORT WriteString, ReadString, Read, WriteLn, WriteCard;
FROM Strings         IMPORT Length;

VAR text     : ARRAY [0..79] OF CHAR;
    ch       : CHAR;
    i,laenge : CARDINAL;

PROCEDURE Codiere(VAR geheim : ARRAY OF BYTE);
CONST kl = 123;
      k2 =  25;
VAR i,n : CARDINAL;
BEGIN
  n := kl;
  FOR i:=0 TO HIGH(geheim) DO
   n := CARDINAL(and(n*k2,255));
   geheim[i] := SHORT(xor(LONG(geheim[i]),n));
  END
END Codiere;

BEGIN
  WriteString("Verschlüsseln und entschlüsseln");
  WriteLn; WriteLn;
  WriteString("Eingabetext: "); ReadString(text);
  laenge := Length(text);
  WriteLn;
  Codiere(text);
  WriteString("Verschlüsselt: "); WriteLn;
  FOR i:= 1 TO laenge DO WriteCard(ORD(text[i]),4) END;
  Codiere(text);
  WriteLn; WriteString("Nun wieder entschlüsselt: "); WriteLn;
  WriteString(text);
  WriteLn; Read(ch)
END VerschluesselnEntschluesseln.
```

3.3 Zugriff auf Systemvariablen

Im Speicherbereich mit den Adressen von 400H bis 512H stehen beim Atari ST die sogenannten Systemvariablen, mit denen das Betriebssystem arbeitet. Sie finden eine ausführliche Liste der Bedeutung dieser Variablen in jedem Buch über das Betriebssystem des Atari ST (z. B. [D],[G]).

Diese System-Variablen liegen in einem geschützten Speicherbereich, auf den nur im Supervisor-Modus (das ist ein bestimmter Zustand des 68000er Prozessors) zugegriffen werden kann. Ihre Programme laufen aber sicherheitshalber im User-Modus ab, wo ein Zugriff auf diesen Bereich einen »Bus error« verursacht. Ansonsten hätte jeder versehentliche Zugriff über einen NIL-Pointer (der meist auf Adresse 0H landet) verheerende Folgen.

Einige Modula-Systeme enthalten in einem Modul `GEMDOS` eine Prozedur `Super`, mit der sich der Supervisor-Modus einschalten läßt. Andere Systeme bieten diese Funktion sicherheitshalber nicht an. Wir bringen Sie daher in einem Modul `SuperVisor`.

```
DEFINITION MODULE SuperVisor;
(* -----------------------------------------------------------------
 * Der Supervisor-Modus sollte nur in Ausnahmefällen eingeschaltet
 * werden, wie zum Beispiel zum Auslesen von Systemvariablen (diese
 * sind nur im Supervisor-Modus lesbar). Er sollte dabei nur so
 * kurz wie unbedingt nötig eingeschaltet bleiben. Danach ist er
 * sobald wie möglich wieder abzuschalten!
 * --------------------------------------------------------------- *)

PROCEDURE SuperVisorEin;          (* Einschalten des Supervisor-Modus *)
PROCEDURE SuperVisorAus;         (* Zurückschalten in den Normalmodus *)
END SuperVisor.
```

Im Implementationsmodul wird die GEMDOS-Funktion Nr. 32 »get/set Supervisor mode« benötigt. Mit TRAP #1 wird GEMDOS aufgerufen, die Funktionsnummer wird zuvor auf den Stack (A7) gelegt.

```
IMPLEMENTATION MODULE SuperVisor;

FROM SYSTEM IMPORT ADDRESS;

VAR
    AlterStack: ADDRESS;             (* Zwischenspeicher für den Stack *)

(*$ L-*)
```

```
PROCEDURE SuperVisorEin;
BEGIN
  ASSEMBLER
        CLR.L     -(A7)              ; Parameter 0: Setze Supervisor-Mode
        MOVE.W    #32,-(A7)          ; GEMDOS-Funktions-Nummer für 'Super'
        TRAP      #1                 ; GEMDOS aufrufen
        ADDQ.L    #6,A7              ; Stack korrigieren
        MOVE.L    D0,AlterStack      ; Vorherigen Supervisor-Stack merken
  END
END SuperVisorEin;

PROCEDURE SuperVisorAus;
BEGIN
  ASSEMBLER
        MOVE.L    AlterStack,-(A7) ; Supervisor-Stack: zum restaurieren
        MOVE.W    #32,-(A7)          ; GEMDOS-Funktions-Nummer für 'Super'
        TRAP      #1                 ; GEMDOS aufrufen
        ADDQ.L    #6,A7              ; Stack Korrigieren
  END
END SuperVisorAus;

(*$ L+*)
END SuperVisor.
```

3.3.1 Bau einer Stoppuhr

Als Anwendung des Moduls `SuperVisor` greifen wir auf die System-Variable »`_hz_200`« zu. Dies entspricht einer LONGCARD-Variablen, die in Abständen von 5 ms (Millisekunden) vom letzten Reset an hochgezählt wird. Damit kann man eine Stoppuhr bauen, um zum Beispiel Laufzeiten von Prozeduren zu messen.

Die Variable »`_hz_200`« steht an der Adresse 4BAH. Die Prozedur `Stoppuhr.Start` liest den augenblicklichen Wert. `Stoppuhr.Lesen` liest ihn zu einen späteren Zeitpunkt und berechnet die Differenz seit dem letzten Aufruf von `Start`, multipliziert sie mit 5 und erhält damit die verstrichene Zeit in Millisekunden. Die Prozedur `Warten` veranlaßt eine Warteschleife, was gelegentlich in Programmen nützlich ist.

```
DEFINITION MODULE Stoppuhr;

PROCEDURE Start;                               (* zum Einschalten der Stoppuhr *)
PROCEDURE Lesen : LONGCARD;           (* gibt Zeit nach dem Einschalten in ms *)
PROCEDURE Warten(Dauer : LONGCARD);                (* Wartet Dauer * 0.005 s *)

END Stoppuhr.
```

Der Implementationsmodul ist auch ganz einfach:

```
IMPLEMENTATION MODULE Stoppuhr;

FROM SuperVisor IMPORT SuperVisorEin, SuperVisorAus;

VAR ZeitTakte    : LONGCARD;
    hz200[4BAH] : LONGCARD; (* System Variable _hz_200, wird alle 5 ms erhöht *)

PROCEDURE Start;
BEGIN
  SuperVisorEin;
  ZeitTakte := hz200;
  SuperVisorAus;
END Start;

PROCEDURE Lesen : LONGCARD;            (* gibt Zeit nach dem Einschalten in ms *)
BEGIN
  SuperVisorEin;
  ZeitTakte := hz200-ZeitTakte;
  SuperVisorAus;
  RETURN ZeitTakte * 5L
END Lesen;

PROCEDURE Warten(Dauer : LONGCARD);
VAR jetzt, spaeter : LONGCARD;
BEGIN
  SuperVisorEin;
  jetzt := hz200;
  spaeter:= jetzt + Dauer;
  REPEAT UNTIL hz200 > spaeter;
 SuperVisorAus;
END Warten;

END Stoppuhr.
```

Hier ein kleiner Testmodul. Vergleichen Sie die gemessenen Zeiten mit einer anderen Stoppuhr.

```
MODULE StoppuhrTest;

FROM InOut   IMPORT Read, WriteLn, WriteString, WriteCard;
IMPORT Stoppuhr;
```

```
VAR taste : CHAR;

BEGIN
  WriteLn; WriteString("Taste drücken für Stoppuhr-Start");
  Read(taste);
  Stoppuhr.Start;
  WriteLn; WriteString("Taste drücken zum Stoppen");
  Read(taste);
  WriteLn; WriteString("Zeit(ms): "); WriteCard(Stoppuhr.Lesen(),1);
  Read(taste);
END StoppuhrTest.
```

3.3.2 Schnelles Zeichnen, direkt auf den Bildschirm

Wir greifen hier das Beispiel `Kreis` (Bresenham-Algorithmus) aus Kapitel 1.3.3 auf. Hier wurde ein Kreis auf dem Textbildschirm gezeichnet oder vielmehr durch das Zeichen »*« angedeutet. Nun schreiben wir – etwas unsauber – direkt in den Bildschirmspeicher; auch das geht in Modula!

Dazu wird die Startadresse des Bildschirms benötigt, also die Adresse desjenigen Bytes, das der linken oberen Ecke des Bildschirms entspricht. Diese Adresse ist in der Systemvariablen »`_v_bas_ad`« gespeichert, die an der Speicherstelle 44EH steht. Der ersten Bildschirmzeile entsprechen dann die 80 Byte, die sich ab der Adresse befinden, die in `_v_bas_ad` steht. 80 Byte entsprechen 640 Bit, die die Pixel einer Bildschirmzeile des monochromen Bildschirms darstellen (0 = weiß, 1 = schwarz). Von diesen 80-Byte-Zeilen haben wir insgesamt 400, die einfach mit aufsteigenden Adressen nacheinander gespeichert sind. Insgesamt sind für den Bildschirm 32000 Byte (400 *80 Byte) reserviert.

Wir benötigen eine Routine `Plot(x,y)`, die ein Pixel auf dem Bildschirm schwarz setzt. Dazu muß sie herausfinden, welches Bit sie in welchem Byte auf eins setzen muß.

Die Umrechnung des Pixels `(x,y)` in die zugehörige Adresse geschieht so:

- `(x,y) → 80y + x DIV 8` (das Byte)
- `7-(x MOD 8)` (das Bit innerhalb des Byte)

In dem gefundenen Byte müssen alle anderen Bits erhalten bleiben, nur das gewünscht Bit soll gesetzt werden. Dies kann man in Modula elegant mit der Inklusion von Mengen lösen. Daher ist es zweckmäßig, den Bildschirm als Feld der Menge `{0..7}` aufzufassen. Das folgende kleine Beispielprogramm zeigt das Zeichnen eines Kreises und einer Linie.

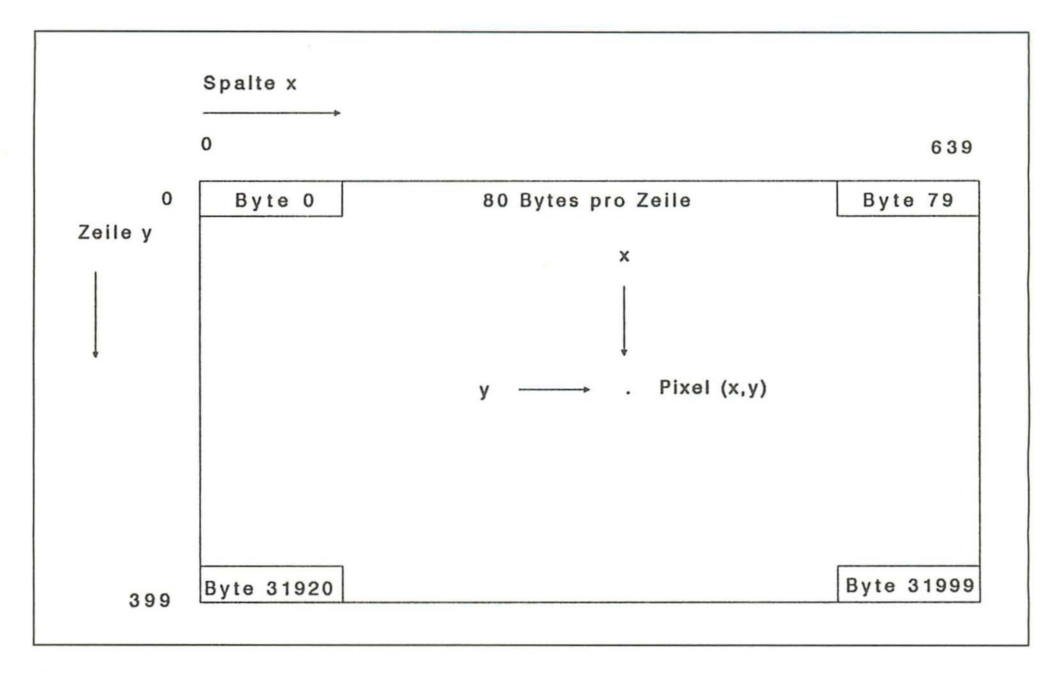


Bild 3.2: Atari-Bildschirm (monochrom)

```
MODULE Grafik;

FROM SuperVisor IMPORT SuperVisorEin, SuperVisorAus;
FROM Terminal   IMPORT Write, Read;

TYPE
    BBS = SET OF [0..7];                                    (* ByteBitSet *)
    BildschirmPtr = POINTER TO ARRAY [0..31999] OF BBS; (* Bildschirm-Array *)

VAR
    Bildschirm : BildschirmPtr;

PROCEDURE HoleBildschirmAdr;
VAR
    VBasAdr[44EH] : BildschirmPtr;      (* Anfang des BildschirmSpeichers     *)
BEGIN
  SuperVisorEin;                        (* 'VBasAdr' nur im SuperVisor lesbar *)
  Bildschirm := VBasAdr;                         (* ... deshalb umkopieren *)
  SuperVisorAus;
END HoleBildschirmAdr;
```

```
PROCEDURE Plot(x,y : INTEGER);
BEGIN
  IF (0 <= x) & Cx <= 639) & (0 <= y) & (y <= 399 ) THEN        (*Clipping *)
    INCL(Bildschirm^[ y * 80 + x DIV 8 ], 7 -(x MOD 8));
  END;
END Plot;

(*$ R- *)    (* Bereichsüberprüfung aus, da r*r > MAX(INTEGER) werden kann *)
PROCEDURE Kreis(xMitte,yMitte,r : INTEGER);
VAR x,y,radiusHoch2 : INTEGER;
BEGIN
  x := 0; y := r; rHoch2 := r * r;
  REPEAT
    Plot(xMitte+x, yMitte+y);
    Plot(xMitte+y, yMitte+x);
    Plot(xMitte+y, yMitte-x);
    Plot(xMitte+x, yMitte-y);
    Plot(xMitte-x, yMitte-y);
    Plot(xMitte-y, yMitte-x);
    Plot(xMitte-y, yMitte+x);
    Plot(xMitte-x, yMitte+y);
    INC(x);
    IF x*x + y*y - y - rHoch2 >= 0 THEN DEC(y) END
  UNTIL x >= y;
END Kreis;

(* $ R+ *)                          (* Bereichsüberprüfung wieder einschalten *)

VAR i : INTEGER;
    c : CHAR;
BEGIN
  Write(33C); Write("E");                               (* Bildschirm löschen *)
  HoleBildschirmAdr;        (* Lesen der Bildschirm-Start-Adresse für 'Plot' *)
  Plot(320,200);
  FOR i:=0 TO 380 BY 20 DO Kreis(320,200,i) END;         (* Kreise zeichnen *)
  Read(c);
  FOR i:=0 TO 399 DO Plot(i,i) END;                         (* Linie zeichnen *)
  Read(c)
END Grafik.
```

Das Zeichnen geschieht sehr schnell. Eine Beschleunigung kann man noch erreichen, wenn die langsamen Operatoren `DIV` und `MOD` durch Prozeduren aus dem Modul `Bitmanipulation` ersetzt werden:

```
x DIV 8 →   shr(x,3)
x MOD 8 →   and(x,7)
```

Wir überlassen dies dem Leser als Übung. Man kann natürlich noch einen Schritt weitergehen und die gesamte Routine `Plot` in Assembler schreiben; das bringt noch eine kleine Geschwindigkeitssteigerung, da die Prozeduraufrufe (für `shr`, `and`) dann wegfallen. Vielleicht packt Sie sogar der Ehrgeiz zur Entwicklung einer eigenen Routine zum schnellen Zeichnen von Linien mit `Plot`. Dies funktioniert ähnlich elegant wie beim Kreisalgorithmus. Man betrachtet wieder die möglichen Nachbarpunkte eines Linienpunkts; das Kriterium für die richtige Auswahl liefert die Steigung.

3.4 Kritisches zur Nutzung von Assembler in Modula-Programmen

Diese Beispiele mögen als Anregung reichen. Vielleicht haben Sie ja Assembler-»Altbestände« in ihrer Software-Sammlung, und können diese nun auch unter Modula verwenden. Wir schließen das Kapitel noch mit einem erhobenen Zeigefinger und knüpfen damit an die eingangs dokumentierte Skepsis an:

1. Einbindung von Assemblerroutinen in Modula-Prozeduren ist prinzipiell nicht nötig, da die flexiblen Jokertypen `BYTE`, `WORD`, `LONGWORD`, sowie `ADDRESS` für systemnahe Konstruktionen zur Verfügung stehen.

2. Für den Aufruf von Betriebssystem-Funktionen gibt es in den meisten Modula-Systemen sprachgerechte Module, die eine Schnittstelle zu diesen Funktionen bereitstellen. Insbesondere zum Laden und Lesen einzelner Register verfügen viele Modula-Systeme über Prozeduren im Modul System.

3. Assemblerroutinen sind nicht portabel auf andere Rechnertypen. Man sollte sie deshalb nur in gesonderten Modulen (»niedrigen Modulen«) benutzen, welche bei Bedarf ausgetauscht werden können. Außerdem empfiehlt es sich, zusätzlich eine Modula-Lösung für diese Routinen bereitzuhalten.

4. Die Erstellung von Assemblerroutinen ist sehr zeitaufwendig. Der erzielte Zeitgewinn beim Programmablauf (von oft nur ein paar Millisekunden) steht in keinem Verhältnis zu dem zusätzlichen Entwicklungsaufwand.

5. Assemblerroutinen sind schlechter wartbar als Hochsprache. Fehler werden schneller gemacht und sind schwer zu erkennen.

Gerade der letzte Punkt hat es in sich:

Fehlerhafte Zuweisungen werden weder vom Compiler erkannt noch vom Laufzeitsystem abgefangen. Ein `MOVE` an die falsche Stelle, eine fehlerhafte Adressierungsart, ein Sprung ins »ungewisse« oder ein nicht richtig abgeglichener Stack führen im besten Fall »nur« zu einem Systemabsturz. Läuft das System dennoch weiter, wird der Fehler eventuell gar nicht erkannt. Inzwischen kann aber die RAM-Disk »zerschossen« worden sein. Noch schlimmer ist, wenn zum Beispiel ein falsches Byte im Harddisk-Buffer landet. Das kann zum kompletten Verlust der Daten führen!

Der einzige Grund, der für den Einsatz von Assemblerroutinen spricht, ist ihre Effizienz:

1. Geschickt programmierte Assemblerroutinen liefern schnelleren Code als die Übersetzung des Modula-Compilers.
2. Der Code ist oft kürzer. Schauen sie doch einmal mit einem Monitorprogramm nach, was der Compiler aus der Prozedur `AusTauschl` vom Anfang des Kapitels macht!

Der zweite Vorteil spielt kaum eine Rolle, da die Laufzeitfunktionen ohnehin den meisten Platz verschwenden. Der erste Vorteil wird nur dann merklich, wenn eine Routine sehr oft gebraucht wird, zum Beispiel bei einem Aufruf in einer Schleife.

Mit den immer besser werdenden Compilern, die den Code eigenständig optimieren, verliert also auch der Vorteil der höheren Effizienz immer mehr an Gewicht. Man sollte bedenken, daß auch schon Betriebssysteme in Modula geschrieben worden sind.

Kapitel 4

Die Programmierung mit Modula-2 unter GEM

Sobald Sie Ihren Atari einschalten, freuen Sie sich über eine grafische Benutzeroberfläche mit Ikonen, Fenstern und Menüs. Dieser Bildschirmzauber ermöglicht der sogenannte GEM (**G**rafics **E**nvironment **M**anager, zu deutsch etwa »Verwalter für die grafische Umgebung«). Sicherlich haben Sie schon längst die bequeme Handhabung beispielsweise einer »File-Selector-Box« zum Anwählen von Dateien schätzen gelernt und wollen nun auch den eigenen Programmen mit diesen Segnungen einen professionellen Charakter geben.

Kein Problem unter Modula! Alle Compiler enthalten ein Paket von GEM-Routinen, die Sie nur in Ihre Programme importieren und aufrufen brauchen. Im Gegensatz zum geplagten Assembler- oder »C«Programmierer findet man je nach Modula-Compiler sogar einige übergeordnete Module vor, die Bereiche dieser Routinen zusammenfassen und die Benutzung von GEM vereinfachen. Beispiele hierzu wären Textwindows unter Megamax und SPC- Modula, oder noch leistungsstärker SSWiS (**S**mall **S**ystems **Wi**ndowing **S**tandard) ebenfalls unter SPC-Modula.

Im einzelnen dürften folgende Themen interessieren:

- Textfenster
- Alertboxen
- File-Selector-Boxen
- Line-A-Grafik
- VDI-Grafik
- Pull-down-Menüs
- Dialogboxen
- SSWiS-Programmierung

Bevor wir jedoch auf die einzelnen Themen zu sprechen kommen, erhalten Sie im ersten Abschnitt einen Überblick über das Betriebssystem des Atari.

4.1 Einführung in die Hierarchie des GEM

Möglicherweise sind Sie von der Vielzahl der zu ihrem Modula-System mitgelieferten GEM-Module erschlagen. Dieser Abschnitt wird Klarheit verschaffen.

Zunächst einmal gibt es beim Atari ein ganz normales Betriebssystem, das sogenannte TOS (»**T**he **O**perating **S**ystem«, im Volksmund auch »**T**ramil **O**perating **S**ystem«). Wie bei den Betriebssystemen CP/M und MS-DOS handelt es sich hierbei um ein Kommando-orientiertes System. Es hat die Aufgabe, zwischen der Benutzeroberfläche und der Hardware zu vermitteln. Es besteht aus dem Trio

- BIOS (**B**asic **I**nput **O**utput **S**ystem)
- XBIOS (E**X**tendet **BIOS**)
- GEMDOS (**GEM** **D**isk **O**perating **S**ystem)

Das BIOS bildet die unterste Ebene der Ein-/Ausgabe. Es überträgt zum Beispiel einzelne Zeichen zwischen Rechner und der Peripherie. Letzteres sind vor allem Bildschirm und Tastatur; auch Drucker, RS232- und MIDI-Schnittstelle gehören dazu.

Das XBIOS ist eine Ansammlung unterschiedlichster Routinen, die das BIOS Atari-spezifisch erweitert: dazu zählt die Kommunikation mit der Maus, dem Bildschirm und den Laufwerken sowie die Bedienung von MIDI- und RS232-Schnittstelle, Tastatur, Timer und vieles andere mehr.

BIOS und XBIOS bilden zusammen den hardware-abhängigen Teil des Betriebssystems. Im Gegensatz dazu stellt GEMDOS den hardware-unabhängigen Teil dar. Zu seinen Aufgaben gehört die Speicher- und Diskettenverwaltung, wobei soviel wie möglich an BIOS und XBIOS delegiert wird. Hier kann der Eindruck entstehen, daß einige Funktionen doppelt vorkommen, also einige Funktionen, die man im BIOS oder im XBIOS sieht, im GEMDOS wieder unter ähnlichem Namen auftauchen. Z. B. schreibt die GEMDOS-Routine `Coconout(ch)` das Zeichen ch auf die »Standard-Ausgabeeinheit« (normalerweise der Bildschirm); der Aufruf der BIOS-Routine `Bconout(device,ch)` bewirkt dasselbe, wenn man als `device` (= Gerät) den Bildschirm angibt.

Tatsächlich handelt es sich aber um ein Zusammenfassen und Übergeben in eine niedrigere Ebene, also das, was man auch im Berufsleben unter Management versteht.

Die GEMDOS-Routinen entsprechen weitgehend denen von MS-DOS und stimmen sogar in den Funktionsnummern überein.

Insgesamt haben wir also mit TOS ein komplettes, konventionelles Betriebssystem. Davon merkt man aber eben nur wenig, denn der Knüller beim Atari ist nun, daß diesem Betriebssystem ein weiterer »Manager« übergeordnet ist, nämlich GEM. Hiermit ist es möglich, dem Computer seine Wünsche nicht wie in einer trockenen Kommandosprache mitzuteilen, sondern in einer benutzerangepaßten »Welt« von Aktenordnern, Fenstern und Papierkorb, die quasi mit dem Zeigefinger (der Maus) bedient werden können. GEM delegiert wiederum die gestellten Aufgaben weitgehend an das TOS. Darüber hinaus gibt es einige Dienste, die TOS nicht erledigen kann; dazu muß GEM direkt auf die Hardware zugreifen.

Im Unterschied etwa zu IBM-Computern, wo man auch GEM zusätzlich laden kann (was zirka 300 Kbyte des ohnehin knappen Speicherplatzes von 640 Kbyte belegt), ist GEM beim Atari fest im ROM eingebaut und belegt zusammen mit dem TOS 192 Kbyte. Insgesamt erhält man folgendes Bild:

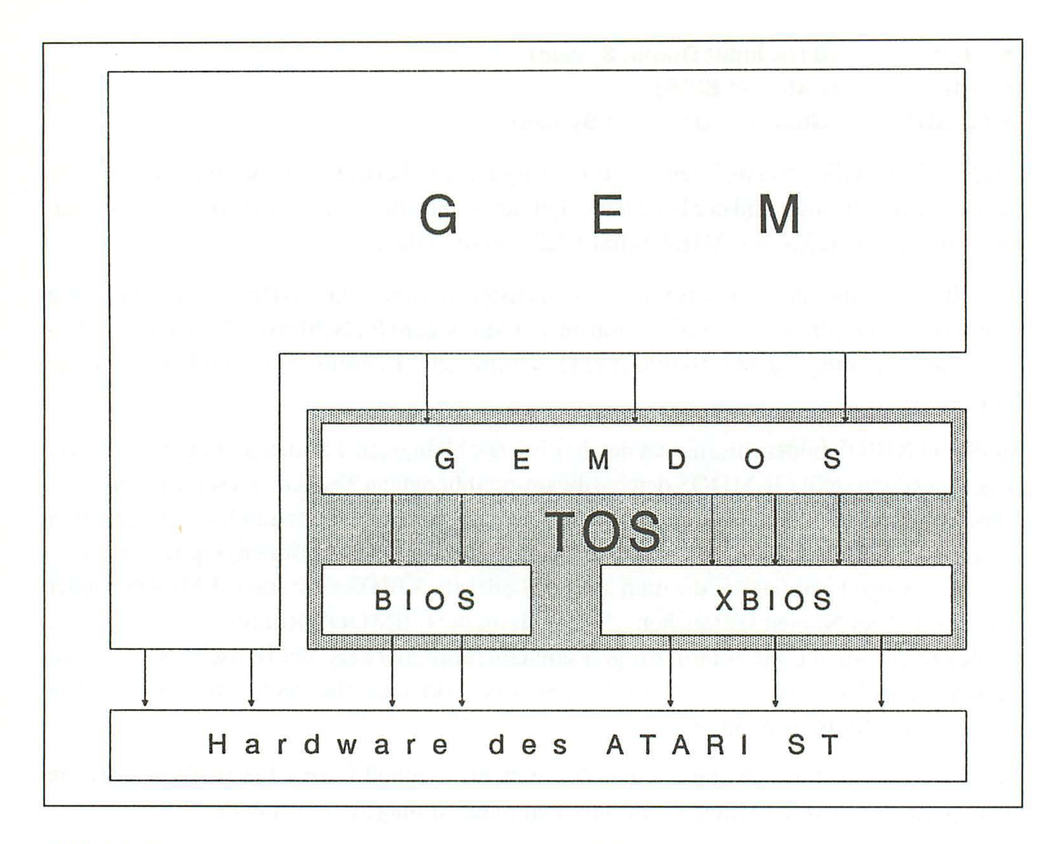


Bild 4.1: Hierarchie des Betriebssystems des Atari-ST

Die Zeichnung macht gleichzeitig die Größenverhältnisse der Betriebssystem-Komponenten deutlich.

GEM gliedert sich zunächst in einen hardware-abhängigen Teil mit 16 sogenannten Line-A-Grafik-Routinen, über die alle Grafik- und Textausgaben des GEM laufen. Diese Routinen arbeiten sehr schnell und sind beispielsweise für die Programmierung von Spielen sinnvoll. Man kann sie natürlich unter Modula nutzen (mehr dazu in Abschnitt 4.5), sie sind aber recht »primitiv«.

Der übergeordnete rechnerunabhängige Teil von GEM unterteilt sich ins AES **A**pplication **E**nvironment **S**ervice) mit 48 und VDI (**V**irtual **D**evice **I**nterface) mit 79 Routinen.

Das VDI stellt eine geräteunabhängige Grafik-Schnittstelle dar und beinhaltet alle Grafikfunktionen, wie Linien und Kreisbögen zeichnen oder Flächen füllen oder Text ausgeben.

Das AES übernimmt die Verwaltung einer Anwendung, die in einer grafisch arbeitenden Umgebung abläuft. Die Palette der AES-Routinen umfaßt umfangreiche Abfragemöglichkeiten für Maus und Tastatur. Auch Fensterverwaltung, die Pull-down-Menüs, Alert- und Dialogboxen fallen hierunter.

Es stellt sich die Frage, wozu der Modula-Programmierer Betriebssystemaufrufe anwenden bzw. warum er den Umgang mit ihnen kennen sollte.

1. Für TOS-Aufrufe sehen wir nur drei Anwendungsbereiche:
   - Lesen von Sondertasten der Tastatur (darauf wurde im Abschnitt 1.7.3 eingegangen).
   - Auslesen des Disketten-Inhaltsverzeichnisses. Wir gehen nicht darauf ein, da einzelne Systeme hierfür fertige Schnittstellen liefern.
   - Erzeugung von Tönen über den eingebauten Soundchip. Hierzu bringen wir im Anschluß einen kleinen Modul.
2. Die Benutzung der Line-A-Routinen wird von den meisten Modula-Systemen unterstützt. Wir gehen darauf im Abschnitt 4.5 ein.
3. Am wichtigsten sind die VDI- und AES-Routinen. Hierzu folgen zahlreiche Beispiele in den Abschnitten 4.6 bis 4.8.

Vor dem Überblick über die VDI- und AES-Routinen gehen wir noch auf die Programmierung des Soundchip YM-2149 als reine TOS-Anwendung ein.

4.1.1 Eine TOS-Anwendung: Programmierung des Soundchip YM-2149

Zur Erzeugung von Tönen, Klängen und Geräuschen bietet der Soundchip 16 Register, R_0 bis R_{15}, deren Bedeutung im folgenden beschrieben wird. Mit der Prozedur `XBIOS.GIWrite` läßt sich unter Megamax-Modula Bytes in diese Register schreiben. Der Modul `XBIOS` liegt bei diesem System in einem speziellen Ordner `TOS`. Sollte Ihr Modula-System `GIWrite` nicht bereitstellen, so finden Sie vermutlich `XBIOS.GIAccess`. Diese Prozedur erlaubt das Beschreiben und Lesen der Soundchip-Register. Beim Schreiben ist der gewünschte Byte-Wert um `080H = 128`$_{\text{dezimal}}$ zu erhöhen.

Zur Tonerzeugung gibt es drei Tongeneratoren (Kanal A, B und C), die gleichzeitig erklingen können! Damit kann man mehrstimmig spielen. Wie veranlaßt man nun einen Tongenerator dazu, einen bestimmten Ton von sich zu geben?

1. Initialisierung: Lautstärke, Tonhöhe usw. einstellen (dazu müssen bestimmte Register geladen werden).
2. Einen (oder mehrere) Kanäle einschalten. Der Ton erklingt solange, bis der Kanal wieder abschaltet. Der Rechner kann derweilen etwas anderes tun, da die 68000-CPU nicht mehr beansprucht wird. Über die CPU werden nur die Register gesetzt, die Tonerzeugung macht der Soundchip eigenständig.
3. Abschalten der eingeschalteten Kanäle, der Ton soll ja nicht ewig klingen.

Die Register des Soundgenerators

Register R_0, R_1:

Das Byte für R_0 sowie die unteren 4 Bit für R_1 bestimmen die Periodendauer des Kanals A. Für eine bestimmte Frequenz (in Hertz) errechnet sich die Periodendauer wie folgt:

```
Periodendauer = 125000 / Frequenz
```

Beispiele: Der höchste Periodenwert ist 4095 (alle 12 Bit auf »1«); ihm entspricht die Frequenz 30.53 Hz. Dem Periodenwert 8 entspricht die Frequenz 15625 Hz. Der Kammerton a' (440 Hz) benötigt die Periodendauer 284.

Register R_2, R_3:

Wie R_0,R_1, jedoch für Kanal B.

Register R_4, R_5:

Wie R_0,R_1, jedoch für Kanal C.

Register R_6:

Die unteren fünf Bits legen die Periodendauer des Rauschgenerators fest.

Register R_7:

Mit diesem Register schaltet man die einzelnen Kanäle ein bzw. aus. Der Rauschgenerator kann zu jedem Kanal zugeschaltet werden. Die Bedeutung der einzelnen Bits in R_7:

Bit Nr.	Bedeutung
0:	0 = Kanal A ein, 1 = Kanal A aus
1:	0 = Kanal B ein, 1 = Kanal B aus
2:	0 = Kanal C ein, 1 = Kanal C aus
3:	0 = Rauschen zu Kanal A zuschalten, 1 = abschalten
4:	0 = Rauschen zu Kanal B zuschalten, 1 = abschalten
5:	0 = Rauschen zu Kanal C zuschalten, 1 = abschalten
6:	0 = Port A als Eingang, 1 = als Ausgang
7:	0 = Port B als Eingang, 1 = als Ausgang

Bit 6 und 7 spielen für den Tongenerator keine Rolle.

Register R_8, R_9, R_{10}:

Die unteren 4 Bit bestimmen die Lautstärke für Kanal A, B bzw. C. Ist Bit 4 (das fünfte) gesetzt, wird die Lautstärke durch einen Hüllkurvengenerator gesteuert.

Register R_{11}, R_{12}:

Die Periodendauer T_H der Hüllkurve wird hiermit festgesetzt (vgl. Abb). R_{11} enthält das untere, R_{12} das obere Byte.

Register R_{13}:

Die unteren 4 Bits definieren die Kurvenform der Hüllkurve. Die Bits heißen:

Bit Nr.	Name
0:	Hold
1:	Alternate
2:	Attack
3:	Continue

Die Auswirkungen entnimmt man der Grafik auf der folgenden Seite.

Register R_{14}, R_{15}:

Spielen für uns keine Rolle.

Der folgende kleine Modul schöpft bei weitem nicht alle Möglichkeiten der Sound-Programmierung aus. Auf die Modula-gerechte Umsetzung mittels des Datentyps Bitset wurde verzichtet. Ebenso auf die Benutzung des Rauschgenerators. Hier ist für den Programmierer ein weites Experimentierfeld gegeben. Wer sich tiefer in die Materie einarbeiten will, dem sei das Buch [M] empfohlen.

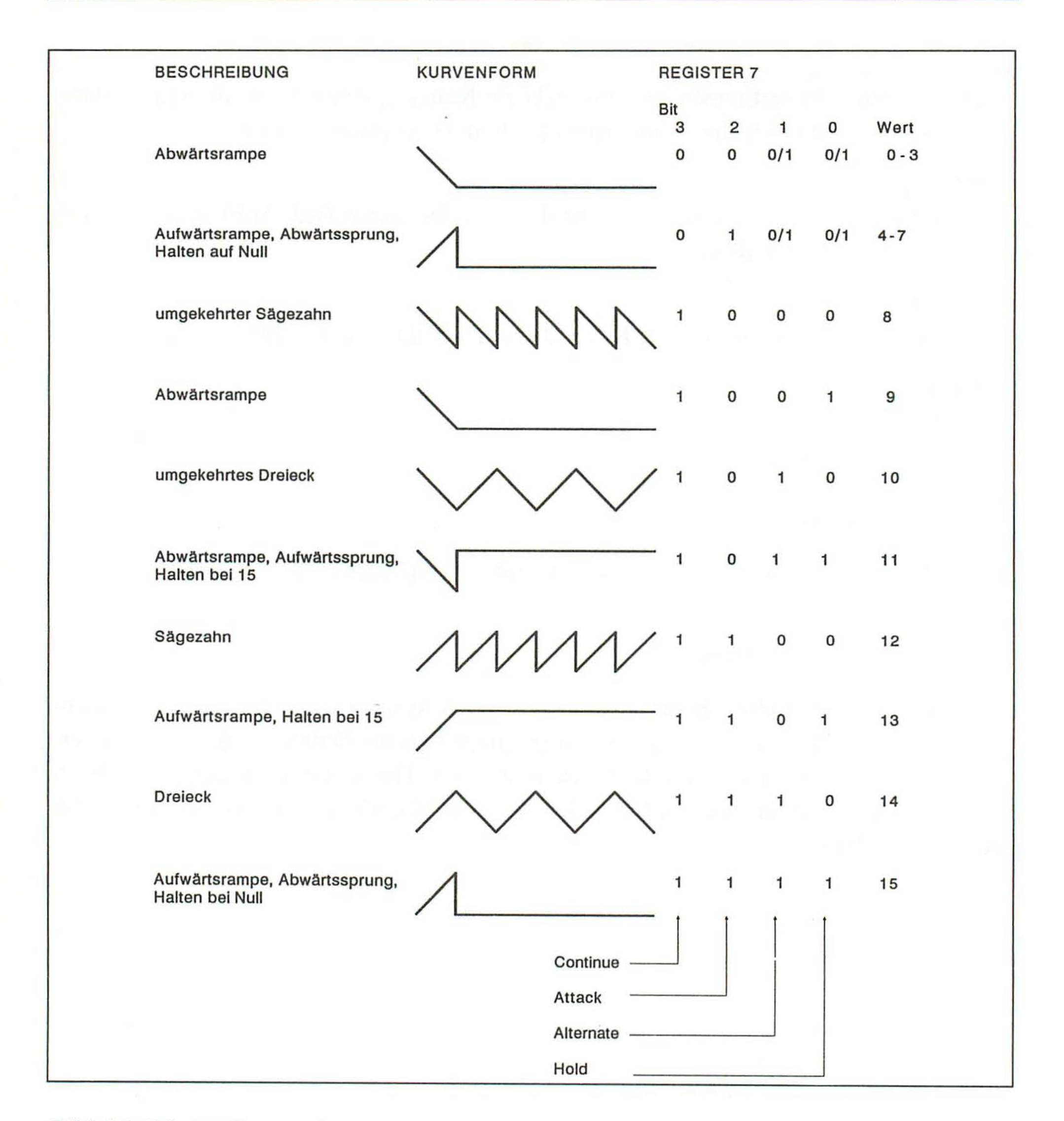


Bild 4.2: Die Hüllkurvenformen

```
DEFINITION MODULE Sound;

PROCEDURE Einstellen(Lautstaerke: CARDINAL);
        (*
         *  Liefert Voreinstellung für den Soundchip YM-2149.
         *  Die Hüllkurven-Periode wird auf einen festen Wert gesetzt
```

```
        * und die Lautstärke (0..15) eingestellt.
        * Bei Lautstärke = 16 wird der Hüllkurven-Generator benutzt.
        *)

PROCEDURE Ton(Periode, Dauer: CARDINAL);
        (*
        * Dient zur Erzeugung eines Tons mit dem Soundchip YM-2149.
        * Frequenz (in Herz) = 125000 / Periode;
        * z.B. Kammerton a' (440 Hz): Periode = 284.
        * 1 <= Dauer <= 255;
        *)

PROCEDURE Aufhoeren;
        (*
        * Zum Abschalten des Soundgenerators.
        * 'Aufhoeren' ist stets am Ende einer Sequenz von Tönen aufzurufen;
        *  anderfalls bleibt der letzte Ton bestehen.
        *)

PROCEDURE Wecker;
        (*
        * Liefert eine Folge von Tönen, um den Programmbenutzer auf
        * etwas hinzuweisen, z.B Beendigung einer Zeichnung.
        *)
END Sound.
```

```
IMPLEMENTATION MODULE Sound;

FROM XBIOS    IMPORT GIWrite;
FROM Stoppuhr IMPORT Warten;

PROCEDURE Einstellen(Lautstaerke: CARDINAL);
CONST HuellkurvenPeriode = 10000;
BEGIN
  GIWrite(8,Lautstaerke);                    (* Reg8  = Lautstärke 0..15   *)
  GIWrite(11, HuellkurvenPeriode MOD 256);
  GIWrite(12, HuellkurvenPeriode DIV 256);
END Einstellen;

PROCEDURE Ton(Periode, Dauer : CARDINAL);
BEGIN
  GIWrite(0,Periode MOD 256);         (* Reg0  = Periodenwert Lowbyte Kanal A *)
  GIWrite(1,Periode DIV 256);         (* Reg1  = Periodenwert Highbyte Kan. A *)
```

```
  GIWrite(7,OFEH);                   (* Reg7  = Schalter,FE = nur Kanal A an *)
                                     (*  Frequenz(in Hertz) = 125000/Periode *)
  GIWrite(13,9);                     (* Reg13 = Hüllkurve: fallende Attack   *)
                                     (*          etwa wie Klavier oder Gitarre *)
  Warten(LONG(Dauer));               (* Warten, bis der Ton abgearbeitet ist *)
END Ton;

PROCEDURE Aufhoeren;
BEGIN
  GIWrite(7,OFFH);                             (* Alle Kanäle ausschalten *)
END Aufhoeren;

PROCEDURE Wecker;
CONST
    c1 = 478; e1 = 369; g1 = 319; c2 = 239;
BEGIN
  Einstellen(16);                                   (* volle Lautstärke *)
  Ton(c1,25); Ton(e1,25); Ton(g1,25);
  Ton(c2,50); Ton(c2,25); Ton(c2,75);
  Aufhoeren                                      (* Soundchip abschalten *)
END Wecker;

END Sound.
```

Probieren Sie den Modul mit dem folgenden kleinen Programm aus:

```
MODULE SoundDemo;

FROM Sound     IMPORT Einstellen, Ton, Aufhoeren, Wecker;
FROM InOut     IMPORT WriteString, WriteLn, ReadCard;
FROM Stoppuhr IMPORT Warten;

CONST
    c1 = 478; d1 = 426; e1 = 369; f1 = 358;
    g1 = 319; a1 = 284; h1 = 253; c2 = 239;

VAR periode, dauer, lautstaerke : CARDINAL;

BEGIN
 Wecker;
 Warten(100);                                      (* 0.5 Sec. warten *)
 Einstellen(15); Ton(4091,255); Aufhoeren;
 Warten(100);                                      (* 0.5 Sec. warten *)
```

```
  Einstellen(16);      (* Lautstärke vom Hüllkurven-Generator regeln lassen *)
  Ton(c1,40); Ton(d1,20); Ton(e1,20); Ton(f1,20);
  Ton(g1,20); Ton(a1,20); Ton(h1,20); Ton(c2,80);
  Aufhoeren;
  Einstellen(15);                                      (* volle Lautstärke *)
  LOOP
    WriteString("Periode (0 = ENDE): "); ReadCard(periode);
    IF periode = 0 THEN EXIT END;
    WriteString("Dauer in 5ms (<256): "); ReadCard(dauer);
    Ton(periode,dauer);
    Aufhoeren
  END;
  Aufhoeren;
END SoundDemo.
```

Nach diesem Beispiel für einen Aufruf einer TOS-Routine wenden wir uns nun den GEM-Routinen zu.

4.1.2 Überblick über die AES- und VDI-Routinen

Die AES- und VDI- Routinen werden von den verschiedenen Modula-Systemen in inhaltlich zusammengehörigen Modulen zusammengefaßt. Dies soll am Beispiel von Megamax-Modula gezeigt werden. Die Nutzung wird in den Programmen der nachfolgenden Abschnitte klargemacht.

Es würde den Rahmen dieses Buches sprengen, alle Prozeduren aus GEM mit der Vielzahl ihrer Parameter hier aufzuzählen. Wir müssen hierzu auf die einschlägige Literatur zum GEM verweisen.

Vielleicht kennen Sie aus C-Programmen die Original-Bezeichnungen der GEM-Routinen. Sie haben oft etwas chaotische Namen wie »v_openvwk«, was davon herrührt, daß der erste C-Compiler nur 8 signifikante Zeichen in einem Bezeichner unterscheidet. Die meisten Modula-Compiler verwenden deshalb eigene Namensgebungen. Hänisch-Modula übernimmt aber die Original-Bezeichner, was dem GEM-erfahrenen Programmierer sicherlich gefallen wird. Hier heißt die Prozedur `v.openvwk`, da der VDI-Modul geschickterweise »v« heißt. Außerdem stimmen die Parameterlisten überein. Auch das MSM2-System hält sich stark an den C-Standard, der im Übrigen von der gesamten Literatur zum GEM üblich ist.

Bei anderen Modula-Compilern stimmen leider Gliederung, Bezeichnung und Parameterlisten der Routinen nicht überein, sie sind aber einander ähnlich. Aus Platzgründen müssen wir für die Abweichung auf die entsprechenden Handbücher verweisen.

Unter Megamax stehen die folgenden Module zur Verfügung:

`VDIAttributes` Attribut-Bibliothek
Voreinstellungen der Art der Ausgabeoperationen für `VDIOutputs`

`VDIControls` Kontrollprozeduren
Löschen des Arbeitsbereiches, Laden von Fonts, Clipping

`VDIEscapes` Escape-Bibliothek
Spezielle Atari-Spezifische Routinen (größtenteils undokumentiert): Hardcopy, Fonts etc.

`VDIInputs` Eingabe-Bibliothek
Eingabe über Tastatur und Maus (Achtung! Einige Routinen funktionieren nicht)

`VDIInquires` Nachfrageprozeduren
Erfragen der Parameter, die mit VDI-Attributes gesetzt worden sind

`VDIOutputs` Ausgabeprozeduren
General Drawing Primitives (GDP, »Grund-Zeichenfunktionen«), Linien, Flächenfüllen usw. (Wichtig!)

`VDIRasters` Raster-Bibliothek
Kopieren beliebiger Rechteckbereiche, Bestimmung der Farbe eines beliebigen Pixels

`AESEvents` Ereignis-Bibliothek
Warten auf Tastatur-, Maus-, oder Timer-Ereignisse

`AESForms` Formular-Bibliothek
Ausführungen von Dialogen (Alertbox, Dialogbox)

`AESGrafics` Grafik-Bibliothek
Verkleinern, Vergrößern, Verschieben von Rahmen auf dem Bildschirm

`AESMenus` Menü-Bibliothek
Bearbeitung von Pull-down-Menüs

`AESMisc` Verschiedens
Unter anderem File-Selector-Box

`AESObjects` Objekt-Bibliothek
Manipulation von sogenannten Objektbäumen (Gruppierungen von Grafik-Objekten)

`AESResources` Resourcebehandlungs-Bibliothek
Bearbeiten von Resourcen, die mit dem Resource-Construction-Set erstellt werden. (z. B. Menüs, Dialogboxen)

`AESWindows` Fenster-Bibliothek
Fenster-Management

Außerdem gibt es bei Megamax noch einige Module, die zur Arbeit mit GEM gehören, wie `GEMEnv`, `GEMGlobals`, `GrafBase`, `LineA`, `ObjHandler`, `EventHandler` und `TextWindows`.

Da die Anwendung von GEM-Aufrufen nicht ganz einfach ist, stellt SPC für fensterorientierte Anwendungen das Modul SSWiS zur Verfügung. Man verspricht sich von SSWiS eine Verbreitung auf andere Systeme (Unix und OS/2-Rechner). Damit wären Modula-Programme, die nur SSWiS für die fensterorientierte Ein-/Ausgabe nutzen, auf solche Rechner portierbar. Außerdem sind die SSWiS-Routinen deutlich einfacher und »ohne lange Vorrede« zu benutzen.

Wichtig ist noch, folgendes zu wissen: Bevor ein Programm GEM-Routinen benutzt, hat es sich höflich beim GEM anzumelden und nach getaner Tat sich wieder abzumelden. Wie man dies macht, zeigen wir in Abschnitt 4.6. Weil hiermit immer wiederkehrende Anweisungen zu erledigen sind, bringen wir hierfür einen externen Modul.

Bei der Benutzung von Line-A-Routinen ist ein Anmelden allerdings nicht erforderlich.

Sicherlich bleiben nach dem erstem Lesen dieses Abschnittes noch etliche Fragen. Diese dürften sich in den nachfolgenden Beispielen klären. Nun endlich hinein ins GEM!

4.2 Benutzung von Textfenstern

Ein Modul »TextWindows« schlägt bereits *N. Wirth* in [W1] als Standardmodul vor. Nahezu alle Modula-Systeme für den Atari bieten eine Implementation. Beim SPC-System stützt sie sich voll auf die SSWiS-Prozeduren.

Eine ähnlich gelungene Implementierung ist die von Megamax-Modula, dem die folgenden Programmbeispiele zugrunde liegen. Die wesentlichen Ein-/Ausgaberoutinen heißen so, wie sie vom Modul `Terminal` bekannt sind (Beispiele: `Goto XY`, `Read`, `Write`, `ReadString`, `WriteString`). Man kann also Programme, die man für die Ein-/Ausgabe auf dem TOS-Bildschirm entwickelt hat, sofort portieren und somit schnell einen einfachen Fensterzauber entfachen. `TextWindows` benutzt die AES-Fensterverwaltungs-Routinen, ohne daß der Programmierer die AES-Module selbst aufrufen muß. Man befindet sich also noch eine Stufe über dem GEM.

Bei jeder Eingabe kann der Benutzer eines mit `TextWindows` geschriebenen Programms die Fenster mit der Maus manipulieren (Verschieben, Größe verändern, mit den Fensterschiebern Scrollen usw.). Lediglich das Schließen des Fensters durch Anklicken des Schließsymbols

muß noch durch das Programm unterstützt werden; dadurch wird sichergestellt, daß das Programm das Schließen zur Kenntnis genommen hat und dort also nichts mehr ausgeben kann.

Das nachfolgende kleine Demonstrations-Programm öffnet zwei Fenster. In jedem Fenster kann man einen String eingeben, der in dem anderen Fenster ausgegeben wird. Spielen Sie ein wenig mit der Maus während des Programmlaufes!

```
MODULE TextWindowDemo;

FROM TextWindows IMPORT Window, Open, Close, Hide, WasClosed,
                        WindowQuality, WQualitySet, ShowMode, ForceMode,
                        GotoXY, ReadString, WriteString, KeyPressed;

VAR Fenster1, Fenster2 : Window;
    ok                 : BOOLEAN;
    s1,s2              : ARRAY[0..50] OF CHAR;

BEGIN
  Open(Fenster1,80,25,WQualitySet{ movable..titled} ,noHideWdw,forceTop,
       "Mein erstes Fenster",10,12,50,10,ok);
  Open(Fenster2,80,25,WQualitySet{ movable..titled} ,noHideWdw,forceTop,
       "Mein zweites Fenster",35,2,40,20,ok);
  WriteString(Fenster1,"Spielen Sie mit der Maus an den Fenstern!" );
  GotoXY(Fenster1,5,5);
  ReadString(Fenster1,s1);
  GotoXY(Fenster2,3,5);
  WriteString(Fenster2,s1);        (* s1 wird in das 2. Fenster übertragen *)
  GotoXY(Fenster2,3,10);
  ReadString(Fenster2,s2);
  GotoXY(Fenster1,5,8);
  WriteString(Fenster1,s2);        (* s2 wird in das 1. Fenster übertragen *)
  REPEAT
    IF WasClosed(Fenster1) THEN Hide(Fenster1) END;  (* Closer bedient? *)
    IF WasClosed(Fenster2) THEN Hide(Fenster2) END;  (* ggfs. schließen *)
  UNTIL KeyPressed();
  Close(Fenster1);
  Close(Fenster2);
END TextWindowDemo.
```

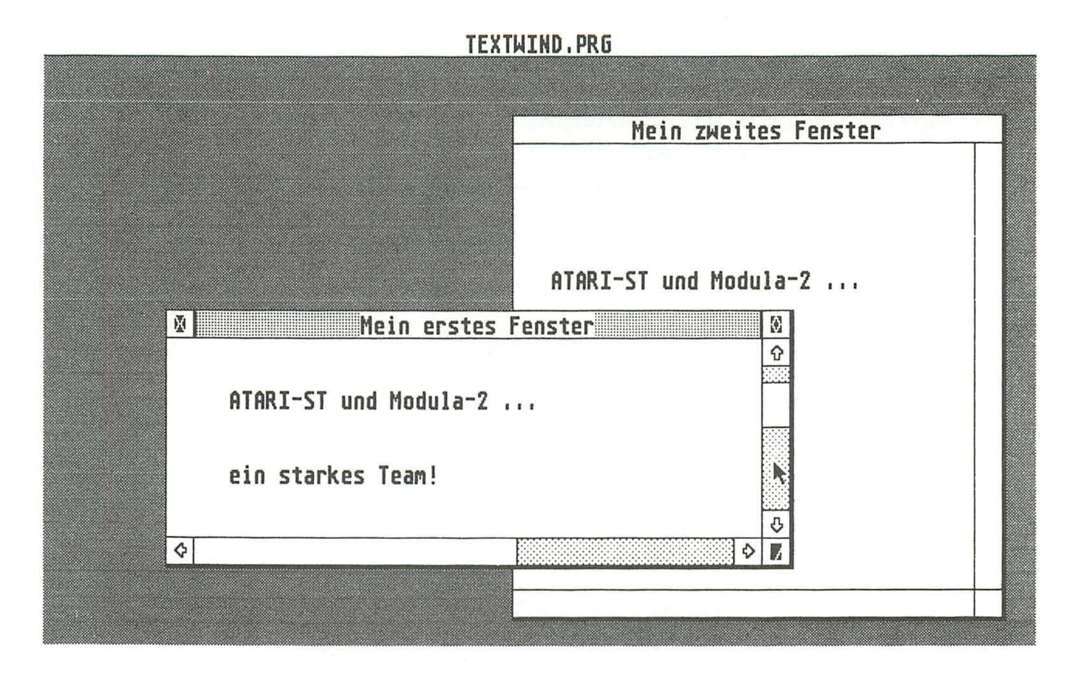


Bild 4.3: Zwei Textfenster

Wenn man keine Grafik benötigt und dennoch mit mehreren Fenstern arbeiten will, empfiehlt sich die Verwendung von `TextWindows`. Die Handhabung ist einfach, das An- und Abmelden beim GEM wird automatisch erledigt. Leider stehen nicht alle komfortablen Eingaberoutinen, die man vom Modul `InOut` kennt, zur Verfügung (es fehlt z.B. `ReadReal`). Will man über mit `TextWindows` erzeugten Fenstern REAL-Zahlen einlesen, so muß man sie erst als String abholen (mit `ReadString`) und dann in REAL-Zahlen umwandeln (dafür gibt es Routinen in `StrConvert`. Da unsere eigenen Leseroutinen aus Abschnitt (1.7.3) auf `InOut` aufbauen, lassen Sie sich sofort für `TextWindows` nutzen, wenn man in der Importliste `InOut` durch `TextWindows` ersetzt.

Eingabesteuerung mit der Maus

Ein besonderer Leckerbissen ist die Benutzung der Maus innerhalb der Fenster. Die Prozedur `DetectChar` übergibt die dem Mauspfeil entsprechende Cursor-Spalte und -Zeile beim Anklicken wieder. Beim folgenden Demonstrations-Programm geht es um das Anklicken von Menüpunkten. Die Menüpunkte stehen in einem Window untereinander; das Programm braucht also nur festzustellen, in welcher Zeile »geklickt« wurde.

```
MODULE TextWindowMitMausDemo;

FROM GrafBase    IMPORT Point, Rectangle;
FROM TextWindows IMPORT Window, Open, Close,
                        WindowQuality, WQualitySet, ShowMode, ForceMode,
                        GotoXY, Write, WriteString, KeyPressed,
                        DetectChar, DetectMode, DetectResult;

CONST
    ESC = 33C;

VAR
    Fenster          : Window;
    ziel             : ARRAY[0..0] OF Window;
    modus            : DetectMode;
    punkt            : Point;
    zeile, spalte,i  : CARDINAL;
    rechteck         : Rectangle;
    ergebnis         : DetectResult;
    ok               : BOOLEAN;
    s                : ARRAY[0..2] OF ARRAY[0..8] OF CHAR;

BEGIN
  Open(Fenster,80,25,WQualitySet{titled},noHideWdw,forceCursor,
       "Fenster mit Mausbedienung",10,3,60,15,ok);
  s[0]:=" Wahl1  "; s[1]:=" Wahl2  "; s[2]:=" Wahl3  " ;
  FOR i:=0 TO 2 DO
    GotoXY(Fenster,10,i+ 5);
    WriteString(Fenster,s[i])
  END;
  ziel[0]:=Fenster;
  modus:=requestPnt;
  REPEAT
    DetectChar(ziel,1,modus,punkt,Fenster,spalte,zeile,rechteck,ergebnis)
  UNTIL (5 <= zeile) & (zeile <= 7);
  GotoXY(Fenster,10,zeile);
  Write(Fenster,ESC); Write(Fenster,"p");  (* VT-52 Steuerzeichen invers *)
  WriteString(Fenster,s[zeile-5]);
  Write(Fenster,ESC); Write(Fenster,"q");  (* VT-52 Steuerzeichen normal *)
  GotoXY(Fenster,10,12);
  WriteString(Fenster,"Sie wählten aus: ");
```

```
    WriteString(Fenster,s[zeile-5]);
    REPEAT UNTIL KeyPressed();
    Close(Fenster);
END TextWindowMitMausDemo.
```

Die Programmierung von »Pull-down«-Menüs wird in Abschnitt 4.7 gezeigt; das Programm »Satellitenbahnen« zeigt eine weitere Anwendung von `TextWindows`.

4.3 Benutzung von Alertboxen

Im vorigen Abschnitt haben wir einen zu GEM übergeordneten Modul benutzt; um das eigentliche Gerangel mit den GEM-Routinen konnten wir uns dabei noch einmal drücken. Nun wollen wir als erstes echtes GEM-Beispiel die einfachste Anwendung vorstellen: die `Alertboxen` (»Alarmkästen«), die man überall in Programmen antrifft, wenn einmal wieder etwas falsch gemacht worden ist. Alles nötige erledigt die Routine `FormAlert` aus `AESForms`. Sie enthält drei Parameter:

1. Parameter:
   die Nummer des vorgewählten Knopfes, der auch mit <Return> zu bedienen ist.

2. Parameter:
   ein String, der die Beschriftungen der Alertbox bestimmt. Er ist folgendermaßen aufgebaut:

   ```
   "[ikonennr][Zeile1|Zeile2...][Knopf1|Knopf2|Knopf3]"
   ```

   Wichtig: es sind nur
   • maximal 5 Zeilen
   • maximal 3 Knöpfe
   erlaubt.

   Die Ikonen-Nummer hat folgende Bedeutung:
   • 0: Kein Icon
   • 1: Ein Icon mit Ausrufezeichen
   • 2: Ein Icon mit Fragezeichen
   • 3: Ein Icon mit Stoppzeichen

3. Parameter:
   Hier steht nach Beendigung der Routine, welchen Knopf der Benutzer angeklickt hat.

Das kleine Programm bringt folgenden Bildschirm:



Bild 4.4: Eine Alertbox

Je nach Knopfdruck erhält man eine entsprechende Antwort. Alles andere geht aus dem Programmtext hervor. Die GEM-An-/Abmeldung erledigen wir mit dem Modul Grafik, der in Abschnitt 4.6 vorgestellt wird.

```
MODULE AlertBoxDemo;

FROM AESForms IMPORT FormAlert;
FROM Grafik IMPORT anmelden, abmelden;

VAR AntwortKnopf : CARDINAL;
    s            : ARRAY [0..80] OF CHAR;

BEGIN
  anmelden;
  s:="[1][Dieses|Modula-Buch|ist|große Klasse][stimmt|weiß nicht|falsch]";
  FormAlert(1,s,AntwortKnopf);
  CASE AntwortKnopf OF
    1 : s:="[1][Gratuliere!|Sie haben|einen guten Geschmack][Ende]" |
    2 : s:="[2][Wir empfehlen Ihnen,|noch etwas weiterzulesen][Ende]" |
    3 : s:="[3][Sie haben|das Buch|wohl nur|kurz durchgeblättert!][Ende]"
```

```
  END;
  FormAlert(1,s,AntwortKnopf);
  abmelden;
END AlertBoxDemo.
```

Das Programmbeispiel wurde mit Megamax-Modula entwickelt. Leider weichen die Implementationen der GEM-Routinen anderer Modula-Compiler leicht voneinander ab.

In SPC-Modula zum Beispiel wird der angeklickte Knopf nicht über einen VAR-Parameter zurückgegeben, sondern als Funktionswert:

```
PROCEDURE Alert(Defaultknopf: INTEGER)
                VAR String: ARRAY OF CHAR): INTEGER;
```

In TDI-Modula findet man eine identische Parameterliste, die Prozedur dafür heißt aber `FormAlert`. Den selben Namen hat sie beim MSM2-System. Hänisch-Modula verwendet den Standardbezeichner `form.alert` (der Modul heißt `form`).

Wir hoffen, daß es dem Leser dieses Buches nicht schwerfällt, die Programmtexte auf seinen eigenen Compiler anzupassen. Der Einheitlichkeit halber halten wir uns im folgenden an die Megamax-Implementation.

4.4 Benutzung einer File-Selector-Box und der Modul »Druck«

Ebenso einfach wie der Umgang mit Alertboxen ist die Programmierung einer File-Selector-Box. Hierzu dient die Prozedur

```
PROCEDURE SelectFile(VAR Pfad,Filename: ARRAY OF CHAR;
                     VAR ok: BOOLEAN);
```

aus `AESMisc`. Bei SPC und TDI-Modula heißt die Prozedur `FileSelectorInput` und stammt aus `AESForms`; bei TDI-Modula ist Pfad und Filename (etwas unhandlich) vom Typ `ADDRESS` und kein VAR-Parameter! Den Standard C-Bezeichner `fsel_input` verwendet das MSM2-System in ähnlicher Weise: hier hießt die Routine `FselInput` (der »_« (Unterstrich) ist in Modula nicht erlaubt). Bei Hänisch-Modula heißt sie `fsel.input`, der Modul heißt raffinierter Weise `fsel`.

Schimpfen Sie also nicht, wenn die Programme auf der mitgelieferten Diskette mit Ihrem System nicht auf Anhieb zu kompilieren sind.

Mit diesem Wissen und dem Handbuch ihres Systems dürfte es Ihnen jedoch jetzt keine Schwierigkeiten mehr bereiten, die Prozeduren entsprechend anzupassen.

Die Prozedur öffnet eine File-Selector-Box entsprechend dem vorgegebenen Pfad (Laufwerk, Ordner). In `ok` steht nach dem Aufruf, wie der Benutzer die Box verlassen hat:

`FALSE`: »Abbruch«-Knopf gedrückt
`TRUE` : »OK«-Knopf gedrückt

In letzterem Fall bestimmen `Pfad` und `Filename` das angewählte File.

```
MODULE FileSelectorBoxTest;

FROM AESMisc IMPORT SelectFile;
FROM Grafik  IMPORT anmelden, abmelden;
FROM InOut   IMPORT GotoXY, WriteString, KeyPressed;

VAR pfad, name : ARRAY [0..80] OF CHAR;
    ok         : BOOLEAN;

BEGIN
  anmelden;
  pfad:="A:*.*"; name:="";
  SelectFile(pfad,name,ok);
  GotoXY(1,23);
  WriteString("Der Pfad heißt: ");
  WriteString(pfad);
  IF ok THEN
    WriteString(", das ausgewählte File heißt: ");
    WriteString(name)
  ELSE WriteString(", es wurde kein File ausgewählt.") END;
  REPEAT UNTIL KeyPressed();
  abmelden;
END FileSelectorBoxTest.
```

Druckprogramm für Textfiles

Wir nutzen die Dienstleistung der File-Selector-Box für ein kleines Druckprogramm. Damit Sie gleich Ihre zahlreichen Modula-Textfiles (sowie andere ASCII-Files) ordentlich formatiert mit Kopfzeile und Seitennumerierung ausdrucken können, liegt das Programm auf der beigefügten Diskette auch in kompilierter Form als ».PRG«-File vor.

Das Programm nimmt über die File-Selector-Box File-Namen entgegen und hängt die Namen der auszudruckenden Files an eine Schlange. Nach Beendigung der Auswahl (wenn der Benutzer »Abbruch« angeklickt hat) werden alle Files nacheinander ausgedruckt und Sie können sich eine Pause gönnen. Das Programm ist während des Druckens mit der <ESC>-Taste abbrechbar. Der Abbruch erfolgt unmittelbar, falls kein Druckerspooler oder sonstiger Zwischenspeicher im Drucker aktiv ist.

```
MODULE Druck;

FROM AESForms IMPORT FormAlert;
FROM AESMisc  IMPORT SelectFile;
FROM Strings  IMPORT String, Length, Delete, Concat, Assign, Copy, Append;
FROM StrConv  IMPORT CardToStr;
FROM Files    IMPORT Create, Open, Close,
                     EOF, State, ResetState, File, Access, ReplaceMode;
FROM Text     IMPORT EOL, Read, Write, WriteString, WriteLn;
FROM Grafik   IMPORT anmelden, abmelden;
FROM GEMDOS   IMPORT PrnOS;

IMPORT InOut;
IMPORT Schlange;

CONST
    ESC  = 33C;
    FF   = 14C;        (* Druckersteuerzeichen für Seitenvorschub (Form Feed)*)
    Fett = 16C;        (* Druckersteuerzeichen für Fettdruck, ggfs. abändern *)
    ZpS  = 64;         (* Anzahl der Zeilen pro Seite - 1 beim Ausdruck      *)
    StandardPfad = "A:\MODULA\*.*";                  (* Standardsuchpfad *)

TYPE
    MeldungsTyp = (anleitung, prnWeg, fileWeg, pause);

VAR
    prn      : File;
    schlange : Schlange.FIFO;
    weiter   : BOOLEAN;

PROCEDURE Meldung(art : MeldungsTyp);
VAR s     : String;
    knopf : CARDINAL;
    ok    : BOOLEAN;
BEGIN
  CASE art OF
    anleitung :
```

```
    s := "[1][Drucker an|Papier ok?|Files wählen|Druckabbruch mit ESC]" |
  prnWeg:
    s := "[3][Drucker|arbeitet nicht!|Bitte prüfen!|Online? Ready?]" |
  fileWeg:
    s := "[3][Das File|ist weg!|Bitte|Diskette prüfen. ]" |
  pause:
    s := "[2][Der Ausdruck|wurde mit|ESC unterbrochen|Was nun ?]" |
 END;
 Append("[Weiter|Abbruch]", s, ok);
 FormAlert(1, s, knopf);
 weiter := (knopf = 1);
END Meldung;

PROCEDURE DruckerTest;
BEGIN
 WHILE NOT PrnOS() AND weiter DO Meldung(prnWeg) END
END DruckerTest;

PROCEDURE FileTest(f: File);
BEGIN
 WHILE (State(f) < 0) AND weiter DO ResetState(f); Meldung(fileWeg) END
END FileTest;

PROCEDURE TastaturTest;
VAR taste : CHAR;
BEGIN
 InOut.BusyRead(taste);
 IF taste = ESC THEN Meldung(pause) END
END TastaturTest;

PROCEDURE DruckerVorbereiten;               (* Die folgenden Einstellungen ....*)
VAR                                         (* ...gelten für Epson Drucker...  *)
BEGIN                                       (* ... nach eigenem Gerät anpassen!*)
 Create(prn,"PRN:",writeSeqTxt,noReplace);                  (* Drucker öffnen *)
 DruckerTest;
 IF weiter THEN
  Write(prn,ESC); Write(prn,"M");                           (* Elite-Schrift  *)
  Write(prn,ESC); Write(prn,"C"); Write(prn,"H");           (* 72 Zeilen/Seite*)
  Write(prn,ESC); Write(prn,"l"); Write(prn,14C);           (* linker Rand 12 *)
  Write(prn,ESC); Write(prn,"R"); Write(prn,0C);            (* am. Zeichensatz*)
 END;
 Close(prn)
END DruckerVorbereiten;
```

```
PROCEDURE WaehleFile(VAR Pfad, DateiName, VollerName: ARRAY OF CHAR): BOOLEAN;
VAR
  i         : CARDINAL;
  FsOk, ok : BOOLEAN;
BEGIN
  SelectFile(Pfad, DateiName, FsOk);
  i := Length(Pfad);
  REPEAT DEC(i) UNTIL Pfad[i] = "\";
  Copy(Pfad, 0, i+ 1, VollerName, ok);
  Append(DateiName, VollerName, ok);
 RETURN FsOk
END WaehleFile;

PROCEDURE FilesAuswaehlen;          (* Wählt alle zu druckenden Files aus...*)
VAR
  FileName, Pfad, FileIdent: String;
BEGIN
  FileName := "";
  Pfad := StandardPfad;                          (* voreingestellter Pfad *)
  WHILE WaehleFile(Pfad, FileName, FileIdent) DO
    InOut.WriteLn; InOut.WriteString(FileName);
    Schlange.Anfuegen(schlange,FileIdent);
  END
END FilesAuswaehlen;

PROCEDURE DruckeEinFile(f: File; Titel: ARRAY OF CHAR);
VAR zeile,seite : CARDINAL;
    SeitenZahl  : String;
    ch          : CHAR;

  PROCEDURE SonderZeichen(nr : CARDINAL);
  (* Anpassung für dt. Zeichen beim Epson FX80, für andere Drucker abändern. *)
  (* Es wird jeweils der dt. Zeichensatz eingeschaltet, das Zeichen gedruckt,*)
  (* und dann  wieder auf den amerikanischen Zeichensatz umgeschaltet.     *)
  BEGIN
    Write(prn,ESC); Write(prn,"R"); Write(prn,2C);     (* dt. Zeichensatz ein *)
    Write(prn,CHR(nr));
    Write(prn,ESC); Write(prn,"R"); Write(prn,0C)      (* am. Zeichensatz ein *)
  END SonderZeichen;

BEGIN
  seite:=1;
  InOut.WriteLn; InOut.WriteString("- Druck von: "); InOut.WriteString(Titel);
  Write(prn,Fett); WriteString(prn, Titel);     (* Kopfzeile ohne Seitennummer*)
  WriteLn(prn); WriteLn(prn);
```

```
    zeile:=3;
    WHILE (NOT EOF(f)) AND weiter DO
      IF zeile MOD ZpS = 0 THEN                              (* Seitenende ? *)
        INC(seite);
        Write(prn,FF);                                          (* Form Feed *)
        Write(prn,Fett);                                        (* Fettdruck *)
        WriteString(prn, Titel);                    (* Kopfzeile mit Seitennr. *)
        WriteString(prn,"                Seite ");
        SeitenZahl:=CardToStr(LONG(seite),0);
        WriteString(prn,SeitenZahl);
        WriteLn(prn); WriteLn(prn);
        zeile:=3;
      END;
      Read(f,ch);
      IF EOL(f) THEN                                          (* Zeilenende? *)
        WriteLn(prn); INC(zeile)
      ELSE
        CASE ch OF
          "ß" : SonderZeichen(126) |
          "ä" : SonderZeichen(123) |
          "ü" : SonderZeichen(125) |
          "ö" : SonderZeichen(124) |
          "Ä" : SonderZeichen( 91) |
          "Ü" : SonderZeichen( 93) |
          "Ö" : SonderZeichen( 92) |
          "§" : SonderZeichen( 64)
          ELSE Write(prn,ch)
        END
      END;
      TastaturTest;
    END;
    Write(prn,FF)                      (* Seitenvorschub für die letzte Seite *)
  END DruckeEinFile;

  PROCEDURE FilesDrucken;
  VAR
    FileIdent, Titel: String;
    i               : CARDINAL;
    ok              : BOOLEAN;
    f               : File;
  BEGIN
    Create(prn,"PRN:", writeSeqTxt, noReplace);
    DruckerTest;
    WHILE NOT Schlange.leer(schlange) AND weiter DO
```

```
      Schlange.Abholen(schlange, FileIdent);
      Open(f, FileIdent, readSeqTxt);
      FileTest(f);
      i := Length(FileIdent);
      REPEAT DEC(i) UNTIL FileIdent[i] = "\";          (* letztes "\" suchen *)
      Copy(FileIdent, i+ 1, 999, Titel, ok);       (* Filenamen herausfischen *)
      IF weiter THEN DruckeEinFile(f, Titel) END;
      Close(f)
    END;
    Close(prn);
  END FilesDrucken;

  VAR c: CHAR;

  BEGIN
    anmelden;
    Meldung(anleitung);
    IF weiter THEN
      Schlange.Einrichten(schlange);
      DruckerVorbereiten;
      IF weiter THEN FilesAuswaehlen; FilesDrucken END;
    END;
    abmelden;
  END Druck.
```

Unser Programm berücksichtigt die Steuerungssequenzen des Epson FX-80 Druckers. Auf der Diskette finden Sie auch eine Version für den Hewlett Packard DeskJet. Außerdem ist hier das Programm leicht geändert, indem ganz auf `InOut` verzichtet wurde, was ein erheblich kürzeres PRG-File bewirkt (Dateineame `DruckNeu`).

Hinweis für SPC-Modula-Programmierer: durch die Benutzung der SPC-Moduls `Printer` läßt sich das Programm stark vereinfachen!

4.5 Benutzung der Line-A-Grafik-Routinen

Grafik läßt sich beim Atari auf zweierlei Weise erzeugen:

1. Mittels der komfortablen GDP-Routinen (Generalized Drawing Primitives) aus `VDI-Outputs`. Darüber mehr im nächsten Abschnitt.

2. Durch die primitiven Line-A-Routinen. Diese sind recht maschinennah und somit sehr schnell. Sie sind allerdings auf den Assembler-Programmierer zugeschnitten, Megamax-Modula bietet eine (relativ) einfache Schnittstelle zu diesen Routinen.

Die Line-A-Routinen werden im Megamax-System durch den Modul `LineA` bereitgestellt. Der Name »Line-A« erklärt sich von der Art und Weise, wie diese Routinen unter Assembler aufgerufen werden: Man setzt an die Stelle, die eine Line-A-Routine aufrufen soll, eine Instruktion, die mit dem Code »`A00`« beginnt. Der 68000er erzeugt dabei einen Trap, daß heißt, er verzweigt dabei zu einer ganz bestimmen Routine. Dabei werden keine Parameter übergeben. Da die Routinen dennoch einiges an Parametern benötigen (Koordinaten, Farb- oder Musterangaben etc.), müssen diese Werte vorher bestimmten Speicherstellen – den »Line-A-Variablen« zugewiesen werden. Einfaches Rezept:

1. Entsprechende Line-A-Variablen vorbelegen (hängt von der benötigten Routine ab)
2. Die Line-A-Routine aufrufen

Ein Beispiel: Man möchte eine durchgezogene Linie vom Punkt (100,200) nach (300,400) ziehen. Das sieht dann so aus:

1. Man besorgt sich einen Zeiger auf die Line-A-Variablen. Das Beispiel in Hänisch-Modula:

```
VAR LineAZeiger: ParamPtr;
    dummy      : ADDRESS;   (* für Parameterliste *)
BEGIN
  LineA.Init(LineAZeiger, dummy);
```

2. Man belegt die erforderlichen Line-A-Variablen wie erwünscht:

```
WITH LineAZeiger^ DO
    x1 := 100;
    x2 := 200;
    x3 := 300;
    x4 := 400;
    LnMask := {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
              (* Alle Bits gesetzt: durchgezogene Linie *)
  END;
```

3. Man ruft `Line` auf (ohne Parameter):

```
LineA.Line;
```

Jetzt ist die Linie auf dem Bildschirm.

Dieses Konzept wird bei Megamax-Modula dadurch aufgeweicht, daß die Prozeduren aus dem Modul `LineA` doch mit einer Parameterliste versehen werden. Mit diesen Parametern belegen sie dann einige (nicht unbedingt alle nötigen) Line-A-Variablen und rufen dann die Line-A-Routine auf. Dies kann durchaus praktisch sein: Zum Beispiel hat hier die Routine `PutPixel` (zeichnet einen Punkt) die Koordinaten und die Farbe mit in der Parameterliste:

```
PutPixel(p:Point;Farbe:CARDINAL);
```

Der Typ `Point` muß aus `GrafBase` importiert werden und enthält die Punktkoordinaten als Verbund.

4.5.1 Der Modul »Line-A-Grafik«

Bei den folgenden Grafik-Programmen, die mit Line-A-Routinen arbeiten, gibt es einige grundlegende Anweisungsfolgen, die immer wieder benötigt werden. Um diese nicht jedesmal neu programmieren zu müssen, sammeln wir diese in Prozeduren und lagern sie in einem externen Modul aus. Ein Beispiel: Bildschirm löschen. Da es dafür keine fertige Routine gibt, realisieren wir dies, indem wir den gesamten Bildschirm mit einem weißem Rechteck übermalen. Damit ein möglicher Bildschirmausdruck besser aussieht, zeichnet man gleich einen Rahmen darum herum.

```
DEFINITION MODULE LineAGrafik;

PROCEDURE LineAHintergrund;
        (*
         *  Löscht den Bildschirm mittels Line-A-Routinen
         *  und setzt das Clipping auf den gesamten Bildschirm.
         *)
END LineAGrafik.
```

```
IMPLEMENTATION MODULE LineAGrafik;

FROM SYSTEM      IMPORT ADR;
FROM GrafBase    IMPORT Pnt, WritingMode;
FROM LineA       IMPORT Line, FilledRectangle,  LineAVariables,
                        PtrLineAVars, HideMouse, ShowMouse;

CONST
    xH        = 639;
    yH        = 399;
```

```
PROCEDURE LineAHintergrund;
CONST
    maxPattern   = 0;          (* Anzahl der 16-Bit Patterns minus 1 *)

VAR
    LineAVarZeiger : PtrLineAVars;
    Muster         : ARRAY[0..maxPattern] OF CARDINAL;

BEGIN
  LineAVarZeiger:=LineAVariables();
  Muster[0]:=0;
  WITH LineAVarZeiger^ DO
    patternMask:=maxPattern;
    patternPtr:=ADR(Muster);
    writingMode:=replaceWrt;
    minClip:=Pnt(0,0);
    maxClip:=Pnt(xH,yH);
    clipping:=TRUE;
  END;
  HideMouse;
  FilledRectangle(Pnt(0,0),Pnt(xH,yH));       (* Bildschirm löschen *)
  WITH LineAVarZeiger^ DO
    plane1:=TRUE; plane2:=TRUE; plane3:=TRUE; plane4:=TRUE;
    lineMask:=MAX(CARDINAL);
  END;
  Line(Pnt(0,0),Pnt(xH,0));                      (* Rahmen zeichnen *)
  Line(Pnt(xH,0),Pnt(xH,yH));
  Line(Pnt(xH,yH),Pnt(0,yH));
  Line(Pnt(0,yH),Pnt(0,0));
  ShowMouse(FALSE);
END LineAHintergrund;

END LineAGrafik.
```

4.5.2 Chaos oder Struktur

Machen wir also endlich einen Punkt (mit `PutPixel`)! Da aber ein einzelner Punkt vielleicht etwas langweilig ist, dürften es auch ein paar mehr sein? Unser Beispielprogramm behandelt das folgende Problem:

Initialisierung:

1. Gegeben sind drei Punkte P1, P2, P3, die nicht auf einer Geraden liegen (wir haben also die Ecken eines Dreiecks).
2. Gegeben ist ein weiterer Punkt, der Startpunkt, »S« (z. B. außerhalb des Dreiecks).

Schleife:

1. Man wähle per Zufallsgenerator einen der Punkte P1, P2, oder P3 aus.
2. Der Mittelpunkt einer gedachten Verbindungslinie von S nach dem ausgewählten Punkt wird auf den Bildschirm gezeichnet. Diese Stelle wird zum neuen Startpunkt S.

Die Schleife läuft solange, bis eine Taste gedrückt wird. Während dieser Zeit werden laufend Punkte nach dem oben beschriebenen Algorithmus gemalt. Das kann doch wohl nur ein Punkte-Chaos werden? Oder besitzt die so entstehende »Punktwolke« doch eine Struktur? Da Sie keine Lust haben, es auf einem Zettel auszuprobieren, lassen Sie das nachfolgende Programm laufen. Programmtechnisch wird hier neben dem Zeichnen von Punkten auch das Einlesen von Koordinaten mit der Maus demonstriert.

```
MODULE ChaosOderStruktur;

FROM Terminal     IMPORT KeyPressed;
FROM RandomGen    IMPORT RandomCard, Randomize;
FROM GEMGlobals   IMPORT MouseButton, MButtonSet, SpecialKeySet;
FROM GrafBase     IMPORT Point;
FROM AESEvents    IMPORT ButtonEvent;
FROM LineA        IMPORT PutPixel, HideMouse, ShowMouse;
FROM LineAGrafik IMPORT LineAHintergrund;

CONST maxEcken        = 3;                    (* Eckenzahl des Vielecks *)

VAR punkte          : ARRAY[1..maxEcken] OF Point;
    start           : Point;
    ZufallsZahl     : [1..maxEcken];
    i               : CARDINAL;

PROCEDURE MausPunkt : Point;          (* liefert einen angeklicken Punkt *)
VAR
    p      : Point;
    knopf  : MButtonSet;
    taste  : SpecialKeySet;
    wieoft : CARDINAL;
```

```
BEGIN
ButtonEvent(1,MButtonSet{msBut1},MButtonSet{msBut1},p,knopf,taste, wieoft);
  RETURN p
END MausPunkt;

PROCEDURE EckenUndStartpunktHolen;
BEGIN                                     (* Alle Ecken mit der Maus holen *)
  HideMouse;
  FOR i:=1 TO maxEcken DO
    ShowMouse(FALSE);
    punkte[i]:=MausPunkt();
    HideMouse;
    PutPixel(punkte[i],1);
  END;
  ShowMouse(FALSE);
  start:=MausPunkt();
  HideMouse;
  PutPixel(start,1);
END EckenUndStartpunktHolen;

PROCEDURE PunkteZeichnen;
BEGIN
  Randomize(1234);                   (* Zufallsgenerator initialisieren *)
  REPEAT
    ZufallsZahl:=RandomCard(1,maxEcken);
    WITH start DO
      x:=(x+punkte[ZufallsZahl].x) DIV 2;
      y:=(y+punkte[ZufallsZahl].y) DIV 2;
    END;
    PutPixel(start,1);
  UNTIL KeyPressed();
  ShowMouse(FALSE);
END PunkteZeichnen;

BEGIN
  LineAHintergrund;
  EckenUndStartpunktHolen;
  PunkteZeichnen
END ChaosOderStruktur.
```

Wetten, daß Sie nicht mit dem folgenden Bild gerechnet haben?

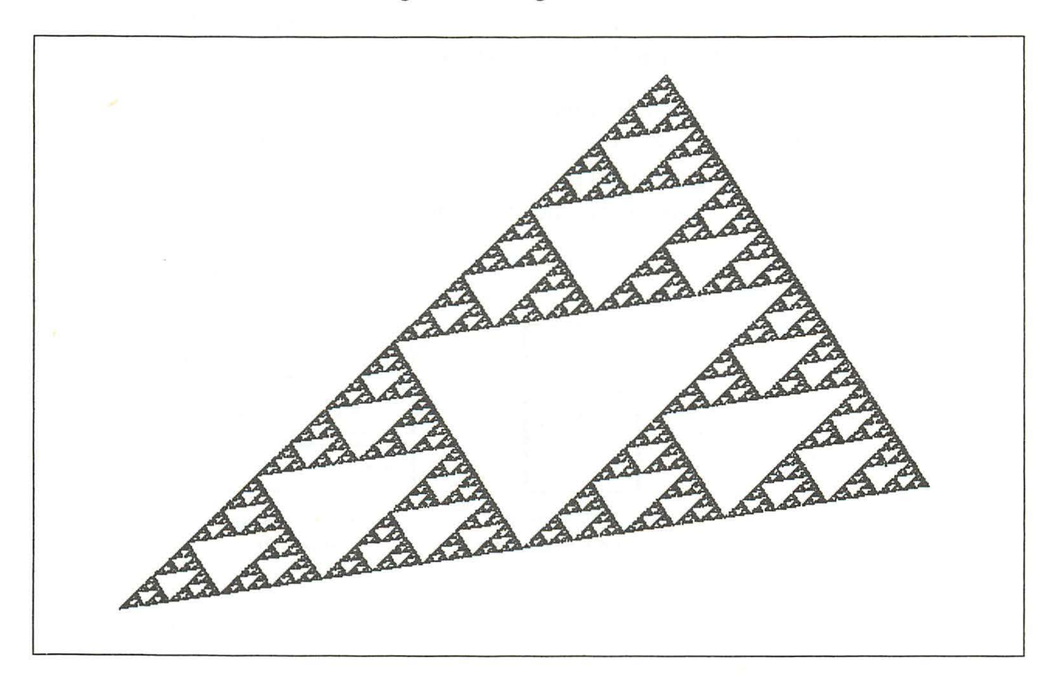


Bild 4.5: Chaos oder Struktur?

Bei genauer Betrachtung erkennt man, daß sich in einigen der hellen Dreiecke ein einzelner schwarzer Punkt befindet (einer, nicht mehr). Wo kommt er her? Wenn es nicht gerade der Startpunkt ist, muß er als Mittelpunkt einer Verbindungsstrecke eines der Eckpunkte P1, P2 oder P3 und einem weiterem Punkt S entstanden sein. Dieser Punkt S liegt dann vom entsprechenden Eckpunkt doppelt so weit weg, wie der Punkt selbst. Er stammt also aus einem doppelt so großem Dreieck, nämlich demjenigen, das von dem Eckpunkt auf die doppelte Entfernung projiziert wurde. Andersherum: wenn ein Punkt in einem (weißen) Dreieck gelandet ist, landet er beim nächsten Mal in einem kleineren. Weiße Dreiecke einer bestimmten Größe werden also nie wieder von einem schwarzen Punkt verschmutzt.

4.5.3 Systemfonts des Atari ST

Eine 2. Demonstration der Line-A-Routinen beschäftigt sich mit der Ausgabe von Texten auf dem Grafik-Bildschirm. Darum einige Vorbemerkungen zur Organisation der Text-Fonts des Atari-GEM:

Zunächst gibt es drei Systemfonts zu 6x6, 8x8 und 8x16 Pixeln. Jeder dieser Fonts enthält 256 Zeichen (der ASCII-Zeichensatz ist darin enthalten). Der Zugriff auf einen der Systemfonts geschieht nun über 2 Indirektionen (Verzeigerungen):

Mit der Funktion `SystemFonts` erhält man einen Zeiger (vom Typ `PtrSysFontHeader`), der auf ein Feld von drei weiteren Zeigern verweist. Jeder dieser Zeiger (Typ: `PtrFontHeader`) verweist auf einen »FontHeader«. Dies ist ein Verbund mit Informationen (unter anderem Größe, Höhe der Kleinbuchstaben) über die drei System-Fonts.

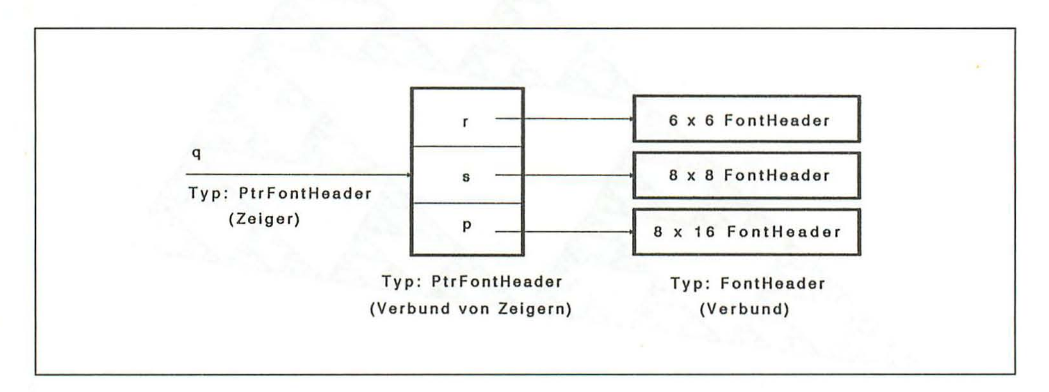


Bild 4.6: Datenstruktur der Atari-Systemfonts

Hat man sich durch diese Verzeigerung durchgehangelt, so kann man nun ein Zeichen aus einem der Fonts mit der Prozedur `TextBlockTransfer` an beliebiger Stelle auf dem Bildschirm ausgeben.

```
MODULE VieleZufallsZeichen;

FROM SYSTEM      IMPORT VAL;
FROM Terminal    IMPORT KeyPressed;
FROM RandomGen   IMPORT RandomCard;
FROM GrafBase    IMPORT Pnt;
FROM LineA       IMPORT TextBlockTransfer, PtrFontHeader, PtrSysFontHeader,
                        SystemFont, SystemFonts, HideMouse, ShowMouse;
FROM LineAGrafik IMPORT LineAHintergrund;

VAR
    p    : PtrSysFontHeader;
    q    : PtrFontHeader;
    ch   : CHAR;
    font : SystemFont;
```

```
BEGIN
  LineAHintergrund;
  HideMouse;
  REPEAT
    p:=SystemFonts();                          (* Zeiger auf Systemfontarray  *)
    font:=VAL(SystemFont,RandomCard(0,2)); (* Zufallsfont auswählen        *)
    q:=p^[font];
    ch:=CHR(RandomCard(33,255));               (* Zeichen zufällig auswählen  *)
                                               (* auf den Bildschirm damit... *)
    TextBlockTransfer(q,ch,Pnt(RandomCard(0,639),RandomCard(0,399)));
  UNTIL KeyPressed();
  ShowMouse(FALSE)
END VieleZufallsZeichen.
```

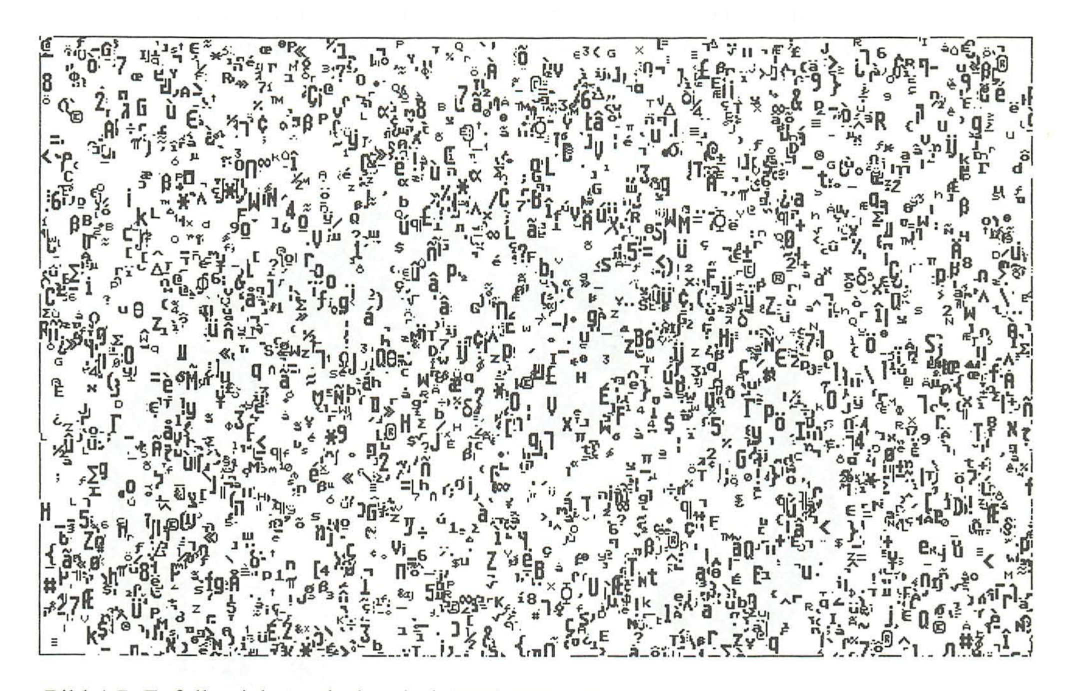


Bild 4.7: Zufallszeichen mit den drei Atari-Systemfonts

4.5.4 Rekursive Grafik

Wunderschöne Grafiken lassen sich durch Rekursion erzeugen. Das Grundprinzip besteht darin, daß eine relativ einfache Figur (etwa ein Dreieck) von einer Prozedur gezeichnet wird, die sich mit veränderten Parametern selbst wieder aufruft und eine verschobene, gedrehte oder gestauchte Figur zeichnet. Bekannte Beispiele sind Hilbert- oder Sierpinski-Kurven. Im Anhang des TDI-Modula-Handbuches findet man hierfür die Beispielprogramme »Sierpinski«, »Diamond« und »Fractal«. Sierpinski-Kurven werden auch in [W1] behandelt; wir greifen diese Beispiele daher nicht auf.

Statt dessen lösen wir eine Aufgabe, die im Rahmen des 5. Bundeswettbewerbs Informatik gestellt wurde:

»Der Baum des Pythagoras setzt sich aus lauter Quadraten zusammen, die so um rechtwinklige Dreiecke angeordnet sind, daß sie den Satz des Pythagoras illustrieren. Ihre Seitenverhältnisse sind 3:4:5. Die ersten Verästelungen des Baumes zeigt die Abbildung.«

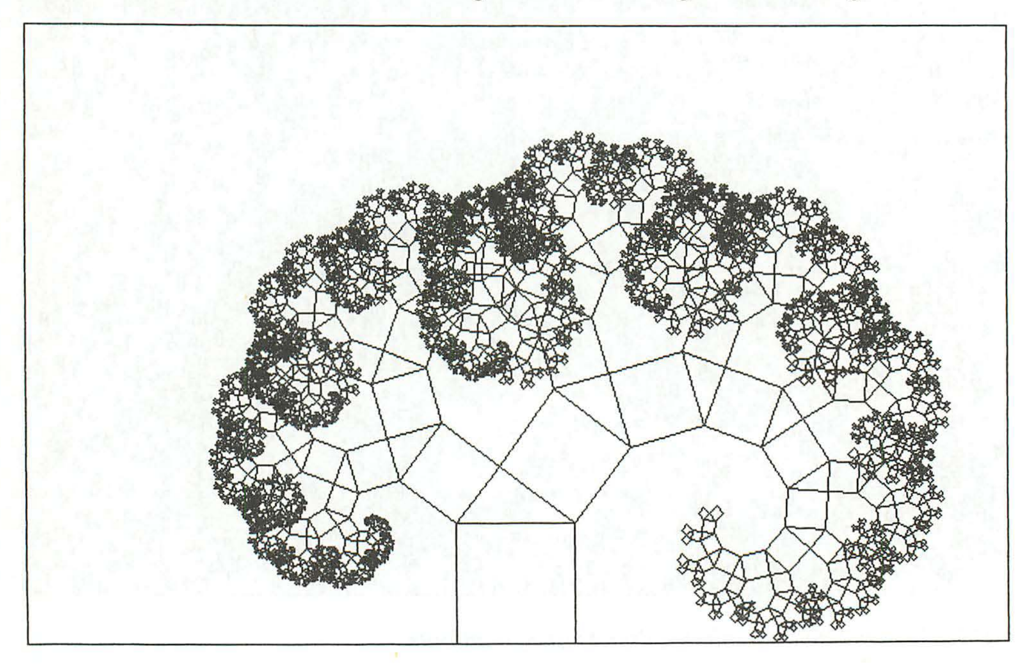


Bild 4.8: Rekursive Grafik: Der Baum des Pythagoras

Programmerläuterung:

Das Programm besteht im wesentlichen aus zwei Prozeduren:

Prozedur `Linie`:

Malt einen Strich (was sonst?). Dabei rechnet sie aber die reellen »Welt«-Koordinaten in Bildschirmkoordinaten um.

Prozedur `ZeichneBaum`:

Malt ein Quadrat (mit `Linie`) und veranlaßt sich selbst, die beiden nächsten Quadrate zu zeichnen.

Die Prozedur `ZeichneBaum` ruft sich zweimal rekursiv auf, einmal für den linken Teilbaum und einmal für den rechten. Wenn man das Programm laufen läßt, sieht man, daß zunächst der äußere rechte Ast fertig gezeichnet wird. Damit nun auch der zweite Aufruf von `ZeichneBaum` für den linken Ast auch einmal ausgeführt wird, muß der erste irgendwann beendet werden. Da sich Baum und jeder Ast aber eigentlich beliebig oft verzweigt, muß irgendwo eine Schranke gesetzt werden. Wir machen das, indem wir die Rekursionstiefe mitzählen: Es wird eine Variable `tiefe` mitgeführt, die bei jedem Aufruf um eins erniedrigt wird. Wenn sie bei `0` gelandet ist, hört die Prozedur auf, sich selbst aufzurufen. Wir beschränken also die Rekursionstiefe.

```
MODULE PythagorasBaum;

FROM Terminal    IMPORT KeyPressed;
FROM MathLib0    IMPORT entier;
FROM GrafBase    IMPORT Pnt;
FROM LineA       IMPORT Line, HideMouse, ShowMouse;
FROM LineAGrafik IMPORT LineAHintergrund;

CONST
   xH              = 639;
   yH              = 399;
   xM              = xH DIV 2;
   RekursionsTiefe = 12;

PROCEDURE Linie(x1,y1, x2,y2 : REAL);
VAR X1,Y1,X2,Y2 : INTEGER;
BEGIN
  X1:=xM +  SHORT(entier(x1));
  Y1:=yH -  SHORT(entier(y1));
  X2:=xM +  SHORT(entier(x2));
  Y2:=yH -  SHORT(entier(y2));
```

```
    Line(Pnt(X1,Y1),Pnt(X2,Y2))
  END Linie;

  PROCEDURE ZeichneBaum(aX,aY,bX,bY : REAL; tiefe : INTEGER);
  VAR cX,cY,dX,dY,eX,ey,lX,lY : REAL;

  BEGIN                                                  (*   .E..     *)
    IF tiefe > 0 THEN                                    (* /        ' *)
      lX:=aY-bY; lY:=bX-aX;                              (* D-------C *)
      cX:=bX+lX; cY:=bY+lY;                              (* |       | *)
      dX:=aX+lX; dY:=aY+lY;                              (* |       | *)
      Linie(aX,aY,bX,bY); Linie(bX,bY,cX,cY);            (* A-------B *)
      Linie(cX,cY,dX,dY); Linie(dX,dY,aX,aY);
      eX:=dX + 0.36*(cX-dX) + 0.48*lX;   ey:=dY + 0.36*(cY-dY) + 0.48*lY;
      ZeichneBaum(eX,ey,cX,cY,tiefe-1); ZeichneBaum(dX,dY,eX,ey,tiefe-1)
    END
  END ZeichneBaum;

BEGIN
  LineAHintergrund;
  HideMouse;

  ZeichneBaum(-FLOAT(xH DIV 16),0.0,FLOAT(xH DIV 16),0.0,RekursionsTiefe);
  REPEAT UNTIL KeyPressed();
  ShowMouse(FALSE)
END PythagorasBaum.
```

4.5.5 Ein Ausflug in die fraktale Geometrie

Zum Schluß noch ein spektakuläres Beispiel: Es geht um die »Mandelbrot-Menge« (benannt nach dem polnischen Mathematiker Benoit B. Mandelbrot) oder besser bekannt unter dem saloppen Namen »Apfelmännchen«.

Wir betrachten eine beliebige komplexe Zahl (vgl. Abschnitt 1.7.3)

```
c = a+bi
```

Dann setzen wir eine komplexe Variable `z := 0` und berechnen fortlaufend

$z := z^2 + c$

Die Mandelbrotmenge ist nun die Menge aller derjenigen Komplexen Zahlen c, für die der Betrag von z nach beliebig häufiger Iteration ($z := z^2 + c$) nicht ins Unendliche wächst. Aus

der Theorie benutzen wir das einfache Resultat, nach dem c nicht zur Mandelbrotmenge gehört, sobald irgendwann einmal |z| > 2 wird.

Da wir allerdings nicht unendlich oft rechnen können, beschließen wir, c schon zur Mandelbrotmenge zu zählen, falls für |z| nach einer bestimmten Zahl von Iterationen (nach etwa 100mal) immer noch |z| < 2 gilt. Markiert man in der komplexen Zahlenebene alle Punkte der Mandelbrotmenge, so ergibt sich eine Figur, das »Apfelmännchen«.

Diese Figur befindet sich im Bereich der Koordinaten [-1.2;1.2] auf der imaginären Achse und [-2.25;0.75] auf der reellen Achse. Besonders einige Stellen am Rande des Apfelmännchens ergeben bei entsprechender Vergrößerung recht interessante Gebilde. Apfelmännchen sind also eigentlich ganz einfach, nur die Umrechnung eines komplexen Punktes in Bildschirm-Koordinaten ist etwas umständlich.

Hier das Programm:

```
MODULE MandelbrotMenge;

FROM Terminal     IMPORT BusyRead;
FROM ComplexLib   IMPORT complex, cmplx, sqrc, addc, abs2;
FROM GrafBase     IMPORT Pnt;
FROM LineA        IMPORT PutPixel, HideMouse, ShowMouse;
FROM LineAGrafik  IMPORT LineAHintergrund;
FROM Sound        IMPORT Wecker;

CONST
    minRe = -2.5;              (* Begrenzungen der komplexen Zahlenebene *)
    minIm = -1.2;
    maxRe =  1.34;           (* Anpassung an das Seitenverhältnis 640/400 *)
    maxIm =  1.2;
    Tiefe =   30;                      (* Iterationstiefe für z:= z^2 + c *)
    xH    =  639;                      (* Bildschirmbegrenzung in Pixeln *)
    yH    =  399;

VAR
    steigungX, steigungY, pX, pY : REAL;
    x,y                          : INTEGER;
    taste                        : CHAR

PROCEDURE rechnePunkt(c : complex) : CARDINAL;
VAR i : CARDINAL;
    z : complex;
BEGIN
  z := cmplx(0.0,0.0);
```

```
  FOR i:=0 TO Tiefe - 1 DO
    z:=addc(sqrc(z),c);                                       (* z:= z^2 + c *)
    IF abs2(z) >= 4.0 THEN RETURN i END
  END;
  RETURN Tiefe
END rechnePunkt;

PROCEDURE Iteration;
BEGIN
  FOR x:=0 TO xH DO
    BusyRead(taste); IF taste = 33C THEN RETURN END;      (* ESC =>Abbruch *)
    pX := minRe + steigungX * FLOAT(x);
    FOR y:=0 TO yH DO
      pY := minIm + steigungY * FLOAT(y);
      IF rechnePunkt(cmplx(pX,pY)) = Tiefe THEN PutPixel(Pnt(x,y),1) END
    END
  END;
END Iteration;

BEGIN
  LineAHintergrund;
  HideMouse;
  steigungX := (maxRe - minRe) / FLOAT(xH + 1);
  steigungY := (maxIm - minIm) / FLOAT(yH + 1);
  Iteration;
  Wecker;
  REPEAT BusyRead(taste) UNTIL taste = 33C;              (* Warten auf ESC *)
  ShowMouse(FALSE)
END MandelbrotMenge.
```

Schon bei der relativ kleinen Iterationstiefe von 30 ergibt sich folgendes Bild, das die Auflösung des Atari-Bildschirmes völlig ausnutzt:

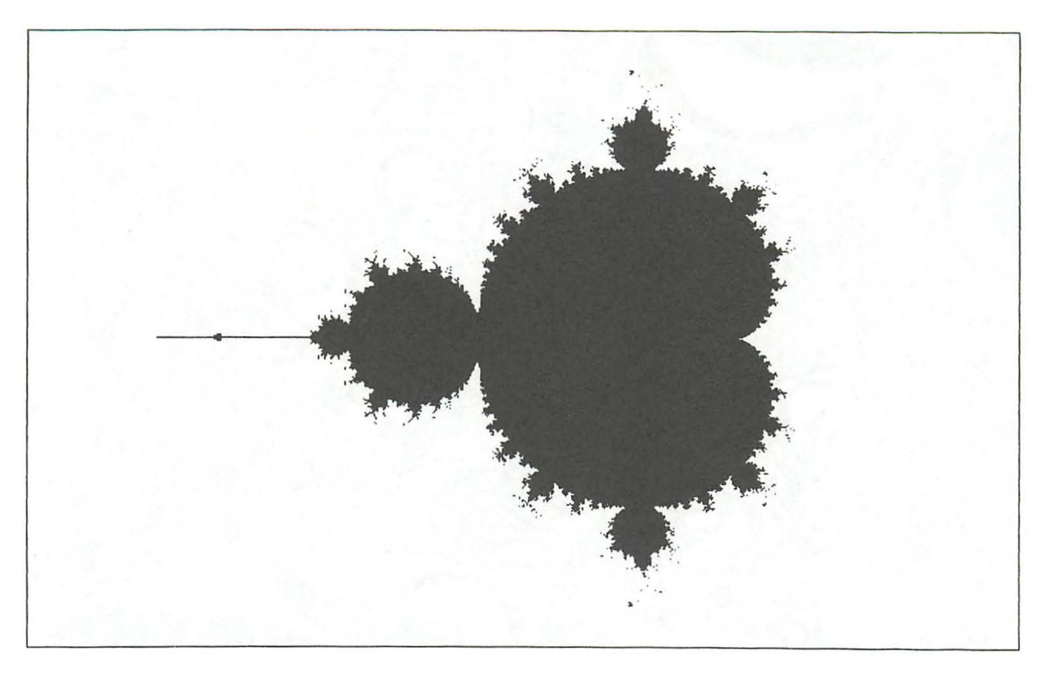


Bild 4.9: Gesamtbild der Mandelbrotmenge

Möchte man die recht verworrenen Randbereiche genauer unter die Lupe nehmen, so braucht man im Programm nur die ersten 4 Konstanten auf den gewünschten Bereich zu setzen. Interessant sind unter anderem folgende Werte:

minRe = -1.254024; maxRe = - 1.252861;
minIm = 0.046252; maxIm = 0.047125;
Tiefe = 200;

Sie finden das so abgewandelte Programm auf der Begleitdiskette unter dem Namen »`MANDELB2.M`«. Die verschiedenen Schwarzweißstufungen haben wir dadurch erreicht, daß nur bei ungradzahliger Tiefe ein Punkt gezeichnet wird. Entsprechend ließen sich für einen Farbmonitor unterschiedliche Farben einsetzen. Der Term `Tiefe MOD FarbAnzahl` bestimmt die Farbe des Bildpunktes.

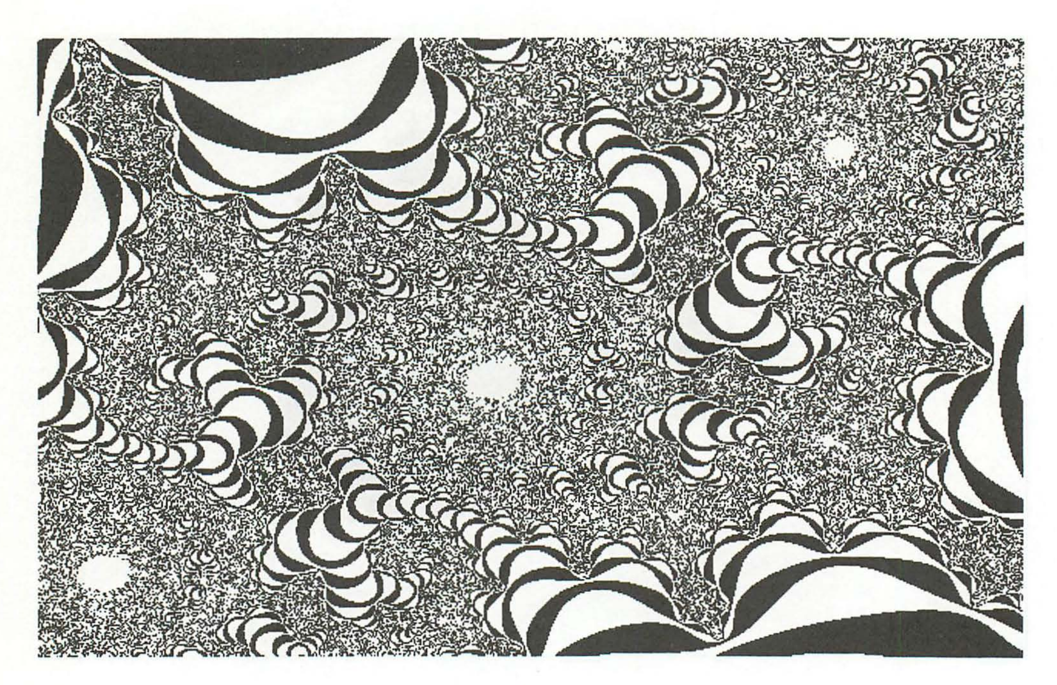


Bild 4.10: Vergrößerter Ausschnitt aus der Mandelbrotmenge (I)

Nur bei angemessener Tiefe erhält man solche eindrucksvollen Grafiken. Man sollte sich also hierfür genügend Zeit lassen und den Rechner ruhig eine Nachtschicht einlegen lassen. Weil es so schön ist, folgt noch ein weiterer Ausschnitt aus der Mandelbrotmenge mit den Grenzen:

minRe = -0.74591; maxRe = - 0.74448;
minIm = 0.11196; maxIm = 0.11339;

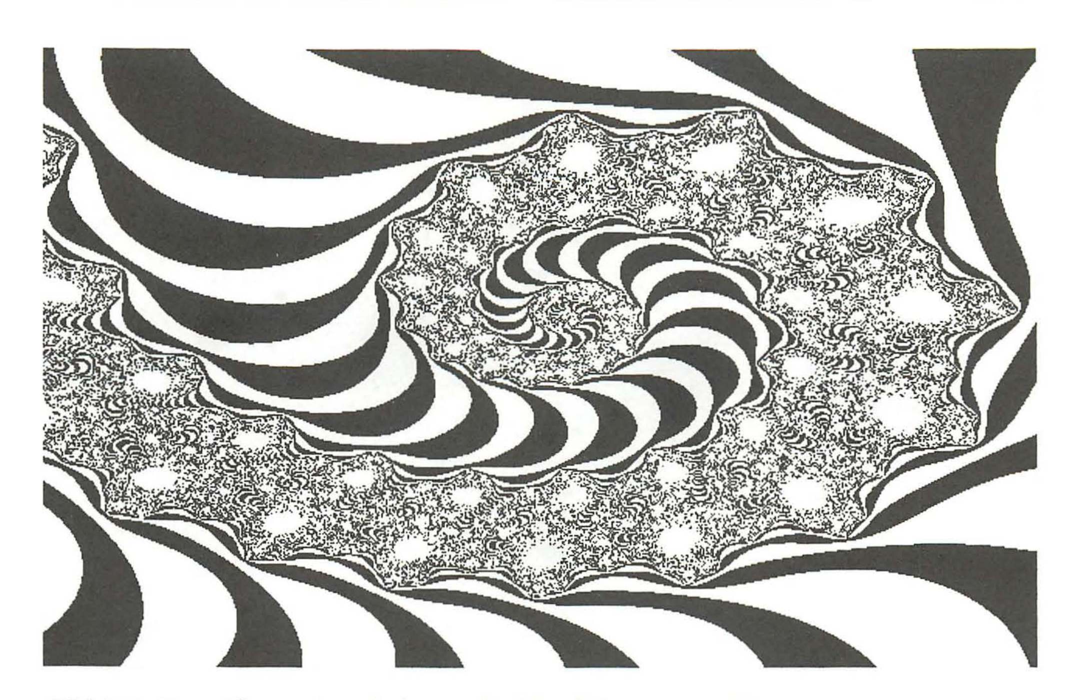


Bild 4.11: Vergrößerter Ausschnitt aus der Mandelbrotmenge (II)

Die Abbildungen zeigen, daß man an die Grenzen der Bildschirmauflösung stößt. Sollten Sie noch anspruchsvoller sein, empfiehlt es sich, die höhere Auflösung eines Druckers auszunutzen. Leiten Sie dazu die Ausgabe auf ein File von Bytes, statt auf den Bildschirm. Lediglich die Routine `PutPixel` muß dazu geändert werden: Ein schwarzer Punkt entspricht einem gesetzten Bit. Nun muß noch ein kleines Ausdrucksprogramm her!

Die Mandelbrotmenge hat viel Rechenzeit auf Rechnern überall in der Welt gekostet, warum sollte es auf Ihrem Atari anders sein?

Juliamengen

Durch eine kleine Modifikation des Mandelbrot-Programmes erhält man völlig andersartige Bilder. Hält man `c` konstant und besetzt `z` mit dem jeweiligen Punkt, so erhält man die »Julia-Mengen« (nach dem französischem Mathematiker *Gaston Julia*). Für `c` wählt man einen Punkt aus der Mandelbrot-Menge.

Für folgende Abbildung haben wir

$c = -0.74543 + 0.11301\, i$

gewählt, einen Punkt im sogenannten »Seepferdchen-Tal«. Änderungen von `c` machen sich durch drastischen Änderung der Julia-Menge bemerkbar.

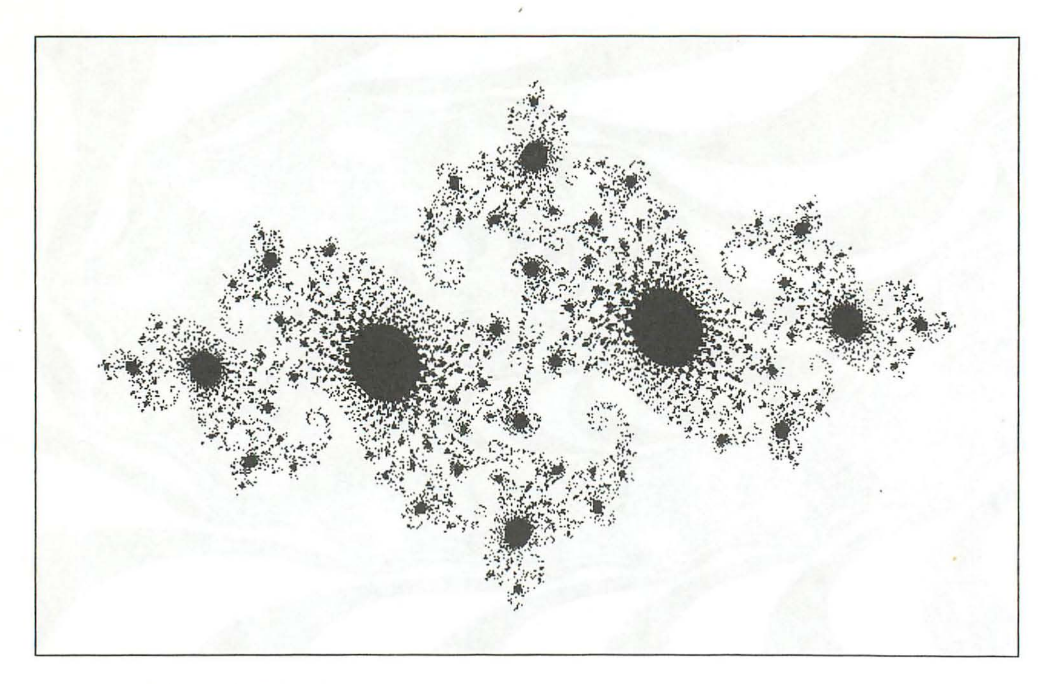


Bild 4.12: Eine Julia-Menge

```
MODULE JuliaMenge;

FROM Terminal     IMPORT KeyPressed;
FROM ComplexLib   IMPORT complex, cmplx, sqrc, addc, abs2;
FROM GrafBase     IMPORT Pnt;
FROM LineA        IMPORT PutPixel, HideMouse, ShowMouse;
FROM LineAGrafik  IMPORT LineAHintergrund;
FROM Sound        IMPORT Wecker;

CONST
    minRe = -1.7;                    (* Begrenzungen der komplexen Zahlenebene *)
    minIm = -1.0;
    maxRe =  1.7;
    maxIm =  1.0;
    Tiefe =  200;                           (* Iterationstiefe für z:= z^2 + c *)
    xH    =  639;                           (* Bildschirmbegrenzung in Pixeln  *)
    yH    =  399;
```

```
VAR
    steigungX, steigungY, pX, pY : REAL;
    x,y                          : INTEGER;
    c                            : complex;

PROCEDURE rechnePunkt2(z : complex) : CARDINAL;
VAR i : CARDINAL;
BEGIN
  FOR i:=0 TO Tiefe DO
    z:=addc(sqrc(z),c);
    IF abs2(z) >= 4.0 THEN RETURN i END
  END;
  RETURN Tiefe
END rechnePunkt2;

PROCEDURE Iteration;
BEGIN
  FOR x:=0 TO xH DO
    IF KeyPressed() THEN RETURN END;
    pX := minRe + steigungX * FLOAT(x);
    FOR y:=0 TO yH DO
      pY := minIm + steigungY * FLOAT(y);
      IF rechnePunkt2(cmplx(pX,pY)) = Tiefe THEN PutPixel(Pnt(x,y),1) END
    END
  END;
  Wecker;
END Iteration;

BEGIN
  LineAHintergrund;
  HideMouse;
  steigungX := (maxRe - minRe) / FLOAT(xH + 1);
  steigungY := (maxIm - minIm) / FLOAT(yH + 1);
  c := cmplx(-0.74543,0.11301);(* Startpunkt am Rand der Mandelbrotmenge, ..*)
                               (* wo sich die beiden großen Gebiete treffen *)
  Iteration;
  REPEAT UNTIL KeyPressed();
  ShowMouse(FALSE)
END JuliaMenge.
```

Einige Vorschläge für die Konstante c zum Experimentieren mit dem Programm:

reell	imaginär	
0	0	im Zentrum der Mandelbrotmenge
0.31	0.04	
-0.11	0.6557	
-0.12	0.74	
0	1	
-0.194	0.6557	sehr schöne Julia-Menge
-1.25	0	
-0.481762	-0.531657	
-0.39054	-0.58679	
-0.15652	-1.03225	
0.11031	-0.67037	
0.27334	0.00742	sehr schöne Julia-Menge

Die Werte sind dem Buch [P] entnommen, was demjenigen empfohlen werden kann, der sich in die mathematische Theorie einarbeiten möchte. Außerdem enthält es hinreißende Farbgrafiken.

Der Mandelbrotmenge und den Julia-Mengen liegen »fraktale« Strukturen zu Grunde. Ihr Rand ist »unendlich stark« zergliedert. Diese diffizile Struktur kommt um so stärker zum Ausdruck, je größer die `Tiefe` gewählt wird. Natürlich erhöht das auch die Rechenzeit. Deutlich schnellere Rechenzeiten lassen sich auf dem Atari nur erreichen, wenn man mit Integer-Arithmetik arbeitet. Der größte Teil der Zeit wird durch für das Rechnen mit REAL-Zahlen verwendet, wobei insbesondere die Multiplikation zu Buche schlägt. Selbst eine Programmierung in Assembler bringt daher wenig.

4.6 Benutzung der VDI-Grafik-Routinen

Mit den 16 Line-A-Routinen, die im vorigen Abschnitt besprochen wurden, lassen sich Punkte, Linien, ausgefüllte Rechtecke und Polygone zeichnen. Sie stellen somit die grundlegenden Ausgabefunktionen auf dem Atari dar, das gesamte GEM benutzt sie (siehe Grafik unter Abschnitt 4.1). Insbesondere stützen sich auch die VDI-GrafikProzeduren auf diese Routinen. Die VDI-Routinen arbeiten daher nicht so schnell, stellen aber komfortablere Nutzungsmöglichkeiten bereit. Sämtliche Beispiele des vorigen Abschnittes sind natürlich auch als VDI-Grafik realisierbar!

Die VDI-Ausgabefunktionen werden durch den Modul `VDIOutputs` bereitgestellt. Dort stehen auch komplexere Funktionen wie zum Beispiel Ausfüllen von Flächen mit diversen Mustern zur Verfügung. Unter anderem gibt 10 »Generalized Drawing Primitives« (im fol-

genden GDP, etwa »allgemeine Zeichen-Grundfunktionen«) zum Zeichnen von einfachen geometrischen Figuren, zum Beispiel: `Circle` (ausgefüllter Kreis), `EllipsPie` (ausgefüllter Ellipsensektor) usw.

Alle Winkelangaben sind ganzzahlig in 1/10tel Grad. Die Füllattribute (Füllmuster), Linienattribute (Strichstärke) und Textattribute (Fontgröße) sind **vor** Benutzung der GDP- Routinen einzustellen.

4.6.1 Der externe Modul »Grafik«

Bevor man anfängt, Anwendungen mit VDI zu programmieren, sollte man sich über eine gemeinsame Schnittmenge von Routinen klar werden, die man in den verschiedenen Programmen benötigt. Diese Prozeduren gehören natürlich »Modula-gerecht« in einen externen Modul `Grafik`. Dadurch läßt sich die Programmierung mit VDI deutlich vereinfachen.

Solche immer wiederkehrenden Probleme sind:

- An- und Abmelden beim GEM
- Bildschirm löschen
- Clipping-Bereiche setzen
- Maustasten abfragen
- Text ausgeben

Mit dem Modul `Grafik` sind diese Aufgaben in einem einfachen Prozeduraufruf zu erledigen. Außerdem kann man, wenn man sich die Quelltexte der Funktionen ansieht, schnell erkennen, welche Schritte ohne diesen Modul nötig wären bzw. nötig sind, sofern man einen eigenen `Grafik`-Modul schreiben will. Unser Modul stellt neben den Prozeduren noch einige Variablen bereit:

`Geraet`:
: Eine Variable vom Typ `DeviceHandle`; sie wird bei jeder VDI-Routine benötigt und steht nach `anmelden` automatisch zur Verfügung

`ArbeitsParm`:
: Ein Verbund, der sich die Koordinaten des aktuellen Arbeitsbereiches merkt. Mit ihm hat der Programmierer, der `Grafik` benutzt, im allgemeinen nichts zu tun.

Eine normale Anwendung dieses Moduls sieht also wie folgt aus:

```
MODULE GrafikDemo;
FROM Grafik IMPORT anmelden, abmelden, Hintergrund, Arbeitsbereich;
<...>
BEGIN
```

```
  anmelden;
  Hintergrund;
  <eigene Anweisungen>
  abmelden
END GrafikDemo.
```

Durch die Einführung dieses Moduls werden die nachfolgenden Programme nicht nur verständlicher, sondern auch unabhängiger vom verwendeten Modula-System. Wenn Sie diesen Modul für Ihr System anpassen, sind auch für nicht-Megamax-Benutzer die Programme leicht übertragbar.

```
DEFINITION MODULE Grafik;
        (*
         * Faßt einige häufig benutzte Grafikgrundfunktionen des VDI
         * zusammen, so daß sie "ohne lange Vorrede" aufgerufen
         * werden können.
         *)

FROM GEMEnv IMPORT DeviceHandle;

VAR Geraet : DeviceHandle;
        (*
         * Das Ausgabegerät ist beim Atari immer der Bildschirm.
         * Sein Device-Handle ( = Kennung) wird für zusätzlich benutzte
         * VDI-Routinen von der folgenden Prozedur "anmelden" bereit-
         * gestellt. "anmelden" muß dazu am Beginn des Programmoduls
         * aufgerufen werden.
         *)

VAR ArbeitsBereich : RECORD
                      xMin, yMin, xMax, yMax: INTEGER
                     END;
        (*
         *  Dies sind die Ausmaße des Grafik-Arbeisbereichs in Bildschirm-
         *  Koordinaten (links,oben) (rechts,unten). Der Verbund wird
         *  durch die Routine "SetzeBereich" initialisiert. Das Anwender-
         *  programm sollte nur lesend darauf zugreifen!
         *)

PROCEDURE anmelden;
        (*
         *  Meldet unsere GEM-Anwendung beim GEM an und
         *  holt sich die Kennung "Geraet".
```

```
        *   Diese Prozedur ist stets vor der Arbeit mit GEM aufzurufen.
        *)

PROCEDURE abmelden;
       (*
        *   Meldet die Anwendung beim GEM ab.
        *   Diese Prozedur ist stets nach der Arbeit mit GEM aufzurufen.
        *)

PROCEDURE Hintergrund;
       (*
        *  Füllt den gesamten Bildschirm mit einem "grauen" Raster.
        *)

PROCEDURE SetzeBereich(x1,y1, x2,y2: INTEGER);
       (*
        *  Füllt ein Rechteck mit der linken, oberen Ecke (x1,y1)
        *  und der rechten unteren Ecke (x2,y2) weiß.
        *  Um das Rechteck wird ein Rahmen und rechts unten
        *  ein Schatten gezeichnet. "Arbeitsbereich" wird initialisiert.
        *  Außerdem wird Clipping für diesen Bereich eingeschaltet.
        *  Damit ist ein Ausgabebereich für Grafik installiert.
        *)

PROCEDURE Plot(x,y: INTEGER);
       (*
        *  Zeichnet den Punkt (x,y).
        *)

PROCEDURE Move(x,y: INTEGER);
       (*
        *  Bewegt den "Zeichenstift" ohne zu Zeichnen nach (x,y).
        *)

PROCEDURE Draw(x,y: INTEGER);
       (*
        *  Zieht eine Linie vom bisherigen Ort zu (x,y).
        *)

PROCEDURE HoleMaus(VAR x,y: INTEGER);
       (*
        *  Diese Prozedur wartet auf das Klicken der linken Maustaste.
        *  Sie übergibt dann die Mauskoordinaten (x,y).
        *)
```

```
PROCEDURE Schreibe(x,y: INTEGER; art: CARDINAL; text: ARRAY OF CHAR);
        (*
         * "text" wird ab (x,y) auf den Bildschirm geschrieben.
         * Bei art=1 wird der 6*6, bei art=2 der 8*8, bei art=3 der
         * 16*8 System-Zeichensatz verwandt; art=4 verwendet einen
         * 30*15 Pixel großen Satz "outlined" (für Überschriften).
         *)

END Grafik.
```

Der Implementationsmodul ist etwas heißer. Wenn sie gleich zu den Grafikanwendungen vorstoßen wollen, sollten sie ihn überschlagen.

In der Prozedur `SetzeBereich` haben wir bewußt keine Fenstertechnik eingesetzt, da Fenster sich nicht in einem Programm mit den Textfenstern aus dem Modul `TextWindows` von Megamax vertragen. Der Hersteller wird diesen Fehler von `TextWindows` aber abstellen.

```
IMPLEMENTATION MODULE Grafik;

FROM GrafBase      IMPORT white, black, Rectangle, Rect, Point, Pnt;
FROM GEMGlobals    IMPORT FillType, TEffectSet, TextEffect,
                          MouseButton, MButtonSet, SpecialKeySet;
FROM GEMEnv        IMPORT RC, DeviceHandle, GemHandle,
                          InitGem, ExitGem, CurrGemHandle;
FROM AESWindows    IMPORT DeskHandle, WindowSize, WSizeMode;
FROM AESEvents     IMPORT ButtonEvent;
FROM AESGraphics   IMPORT GrafMouse, arrow;
FROM VDIControls   IMPORT SetClipping;
FROM VDIOutputs    IMPORT GrafText, FillRectangle, Line, PolyLine;
FROM VDIAttributes IMPORT SetFillType, SetFillColor, SetFillIndex,
                          SetTextColor, SetTextEffects, SetAbsTHeight;
FROM VDIInputs     IMPORT HideCursor, ShowCursor;

VAR
    GemKennung   : GemHandle;      (* wird zum An- und Abmelden benötigt *)
    AltesX,AltesY: INTEGER;

      (* ============ Initialisierungs-Prozeduren ============ *)

PROCEDURE anmelden;
VAR ok: BOOLEAN;
BEGIN
  InitGem(RC, Geraet, ok);         (* Unser Programm beim GEM anmelden    *)
```

```
  IF NOT ok THEN HALT END;             (* GEM will nicht => brutal Abbrechen *)
  GemKennung := CurrGemHandle();
  GrafMouse (arrow, NIL);
END anmelden;

PROCEDURE abmelden;
BEGIN
  ExitGem(GemKennung)
END abmelden;

PROCEDURE Hintergrund;
BEGIN                    (* Es wird immer vorausgesetzt, daß die Maus da ist *)
  HideCursor(Geraet);                (* Maus immer beim Zeichnen verstecken *)
  SetFillIndex(Geraet, 4);                              (* Füllmuster waehlen *)
  SetFillType(Geraet, dottPattern);
  FillRectangle(Geraet, WindowSize(DeskHandle, borderSize));       (* alles *)
  ShowCursor(Geraet,FALSE);
END Hintergrund;

PROCEDURE SetzeBereich(x1,y1, x2,y2: INTEGER);
VAR bereich  : Rectangle;
    EckPunkte: ARRAY [0..4] OF Point;
    i        : INTEGER;

BEGIN
  WITH ArbeitsBereich DO
    xMin := x1; yMin := y1; xMax := x2; yMax := y2
  END;
  bereich := Rect(x1, y1, x2-x1+ 1, y2-y1+1 );
  HideCursor(Geraet);
  SetFillColor(Geraet, white);
  SetFillType(Geraet, solidFill);
  SetClipping(Geraet,WindowSize(DeskHandle, borderSize));
  FillRectangle(Geraet, bereich);                       (* Fläche weiß füllen *)
  EckPunkte[0]:=Pnt(x1,y1); EckPunkte[1]:=Pnt(x2,y1);     (* für den Rahmen *)
  EckPunkte[2]:=Pnt(x2,y2); EckPunkte[3]:=Pnt(x1,y2);
  EckPunkte[4]:=EckPunkte[0];
  PolyLine(Geraet,EckPunkte,4);                          (* Rahmen zeichnen *)
  INC(x1,2); INC(y1,2);                                  (* für den Schatten *)
  FOR i:=1 TO 3 DO
    EckPunkte[0]:=Pnt(x2+i,y1); EckPunkte[1]:=Pnt(x2+i,y2+i);
    EckPunkte[2]:=Pnt(x1,y2+i);
    PolyLine(Geraet,EckPunkte,2)
  END;
```

```
  SetClipping(Geraet, bereich);
  ShowCursor(Geraet,FALSE)
END SetzeBereich;

     (* ========== Grundlegende Grafik-Prozeduren =========== *)

PROCEDURE Plot(x,y: INTEGER);
BEGIN
  Line(Geraet, Pnt(x,y), Pnt(x,y))
END Plot;

PROCEDURE Move(x,y: INTEGER);
BEGIN
  AltesX := x;
  AltesY := y
END Move;

PROCEDURE Draw(x,y: INTEGER);
BEGIN
  Line(Geraet, Pnt(AltesX,AltesY), Pnt(x,y));
  AltesX := x;
  AltesY := y
END Draw;

     (* ============ Maus und Text in der Grafik ============ *)

PROCEDURE HoleMaus(VAR x,y: INTEGER);
VAR
    p     : Point;
    knopf : MButtonSet;
    taste : SpecialKeySet;
    wieoft: CARDINAL;
BEGIN
  ButtonEvent(1,MButtonSet{msBut1,msBut1},MButtonSet{msBut1,msBut2},
              p,knopf,taste,wieoft);
  IF knopf = MButtonSet{msBut1} THEN  x := p.x; y := p.y END;
END HoleMaus;

PROCEDURE Schreibe(x,y: INTEGER; art : CARDINAL; text: ARRAY OF CHAR);
VAR dummy, hoehe : CARDINAL;
BEGIN
  HideCursor(Geraet);
  SetTextColor(Geraet, black);
  SetTextEffects(Geraet, TEffectSet{});                 (* Normalschrift *)
```

```
  CASE art OF
    1 : hoehe:=4 |                                          (* 6*6 Fond *)
    2 : hoehe:=6;                                           (* 8*8 Fond *)
    3 : hoehe:=13 |                                        (* 8*16 Fond *)
    4 : hoehe:=25;SetTextEffects(Geraet,TEffectSet{outlineText})  (*15*30*)
  ELSE hoehe:=13 END;
  SetAbsTHeight(Geraet,hoehe,dummy,dummy,dummy,dummy);
  GrafText(Geraet, Pnt(x,y), text);
  ShowCursor(Geraet,FALSE)
END Schreibe;

END Grafik.
```

4.6.2 Erstellung von Tortendiagrammen

Als erste VDI-Grafik-Anwendung zeigen wir eine Tortengrafik (auch »Kreisdiagramm«). Der Prozedur `Torte` wird dazu der Mittelpunkt und der Radius des Kreises übergeben, sowie ein offenes Feld von Werten einer beliebigen Statistik. Die Prozedur rechnet diese Werte in die entsprechenden Winkelmaße um und benutzt die VDI-Routine `Pie` zum Zeichnen eines ausgefüllten Kreissegmentes. Die Zeichnung beginnt oben und füllt die Segmente im Kreisdiagramm mit jeweils einem anderem Füllmuster. Die Winkelmaße werden in Zehntel Grad gemessen. Anfangs wird für 90° äquivalent 450° genommen, damit die Winkel beim Subtrahieren positiv bleiben.

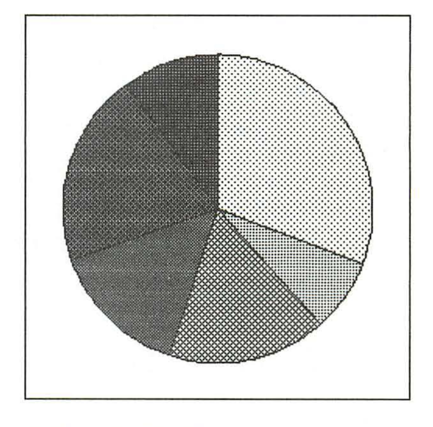


Bild 4.13: Ein Tortendiagramm

```
MODULE TortenGrafik;

FROM VDIAttributes IMPORT SetFillType, SetFillIndex, SetFillColor,
                          SetFillPerimeter;
FROM AESEvents     IMPORT KeyboardEvent;
FROM VDIInputs     IMPORT HideCursor, ShowCursor;
FROM VDIOutputs    IMPORT ElliptPie;
FROM GEMGlobals    IMPORT FillType, GemChar;
FROM GrafBase      IMPORT Point,Pnt,black;
FROM Grafik        IMPORT anmelden, abmelden, Geraet, Schreibe,
                          Hintergrund, SetzeBereich;
```

```
PROCEDURE Torte(mitte : Point; radius : CARDINAL; werte : ARRAY OF REAL);
VAR
    i, AltWinkel, NeuWinkel, yradius : CARDINAL;
    summe, akku                      : REAL;
BEGIN
  summe := 0.0;
  FOR i := 0 TO HIGH(werte) DO summe := summe + werte[i] END;
  SetFillType(Geraet, dottPattern);          (* schwarzen Rand voreinstellen *)
  SetFillColor(Geraet, black);
  SetFillPerimeter(Geraet, TRUE);
  HideCursor(Geraet);                            (* Maus stört beim Zeichnen *)
  akku := 0.0; AltWinkel := 0;
  yradius := radius * 9 DIV 10;     (* zum Ausgleich der Verzerrung --> Kreis *)
  FOR i := 0 TO HIGH(werte)  DO
    SetFillIndex(Geraet, i MOD 24 + 1);                 (* Füllmuster setzen *)
    akku := akku + werte[i];
    NeuWinkel := SHORT(TRUNC(akku * 3600.0 / summe));
    EllipttPie(Geraet,mitte,radius,yradius, 4500 - NeuWinkel, 4500 - AltWinkel);
    AltWinkel := NeuWinkel
  END;
  ShowCursor(Geraet, FALSE)
END Torte;

VAR
    Testwerte : ARRAY [0..5] OF REAL;
    dummy     : GemChar;

BEGIN
  Testwerte[0] := 55.5;
  Testwerte[1] := 13.7;
  Testwerte[2] := 30.0;
  Testwerte[3] := 25.9;
  Testwerte[4] := 36.8;
  Testwerte[5] := 18.1;
  anmelden;
  Hintergrund;
  SetzeBereich(9,9, 630, 390);
  Schreibe(184,50,3, "Demonstration einer Tortengrafik");
  Torte(Pnt(300,200), 100, Testwerte);
  KeyboardEvent(dummy);
  abmelden;
END TortenGrafik.
```

4.6.3 VDI-Grafik-Textausgabe

Mit der Routine »GrafText« läßt sich Text in VDI-Grafiken ausgeben. Sie baut auf den Systemfonts auf, die in den Größen 6x6, 8x8 und 8x16 Pixel vorhanden sind. Es ist aber durch Setzen der »Absoluten Textgröße« eine Vergrößerung möglich. Einen Überblick gibt die folgende Bildschirmkopie.

```
Zeichensatz AbsH:  4,  6 *  6
Zeichensatz AbsH:  5,  7 *  7
Zeichensatz AbsH:  6,  8 *  8
Zeichensatz AbsH:  7,  9 *  9
Zeichensatz AbsH:  8, 11 * 10
Zeichensatz AbsH:  9, 12 * 11
Zeichensatz AbsH: 10, 13 * 13
Zeichensatz AbsH: 11, 15 * 14
Zeichensatz AbsH: 13,  8 * 16
Zeichensatz AbsH: 14,  9 * 17
Zeichensatz AbsH: 15,  9 * 18
Zeichensatz AbsH: 16, 10 * 19
Zeichensatz AbsH: 17, 10 * 21
Zeichensatz AbsH: 18, 11 * 22
Zeichensatz AbsH: 19, 12 * 23
Zeichensatz AbsH: 20, 12 * 24
Zeichensatz AbsH: 21, 13 * 25
Zeichensatz AbsH: 22, 14 * 26
Zeichensatz AbsH: 23, 14 * 28
Zeichensatz AbsH: 24, 15 * 29
Zeichensatz AbsH: 25, 15 * 30
```

Bild 4.14: Grafikzeichensätze

Man sieht, daß nur die Texte mit der eingebauten Fontgröße gut aussehen. Die übrigen Größen werden aus diesen durch Umrechnen erzeugt, dadurch entstehen bei ungünstigen Verhältnissen zuweilen deutliche Verzerrungen. Dummerweise »hängt« sich der Atari bei den absoluten Texthöhen 12 und 26 auf. Gerade diese Werte würden keine Verzerrungen erwarten lassen, da sie ein Vielfaches der Systemfont-Größen darstellen! Die »Absolute Höhe« 12 ist die Verdopplung des 8x8 Fonts, 26 die des 8x16 Fonts.

Mit dem nachfolgendem kleinen Programm können Sie sich die Zeichensätze näher ansehen:

```
MODULE GrafikZeichenSaetze;

FROM Terminal       IMPORT KeyPressed;
FROM Strings        IMPORT String, Append;
FROM StrConv        IMPORT CardToStr;
FROM GrafBase       IMPORT Pnt;
FROM VDIOutputs     IMPORT GrafText;
FROM VDIAttributes IMPORT SetAbsTHeight;
FROM Grafik         IMPORT Geraet, anmelden, abmelden, SetzeBereich;

VAR
    AbsHoehe, ChWeite, ChHoehe, ZeWeite, ZeHoehe : CARDINAL;
    s, s1, s2, s3, s4, s5                        : String;
    y                                            : INTEGER;
    ok                                           : BOOLEAN;

BEGIN
  anmelden;
  SetzeBereich(0,0,639,399);
  y:=10; AbsHoehe:=4;
  REPEAT
    SetAbsTHeight(Geraet,AbsHoehe,ChWeite,ChHoehe,ZeWeite,ZeHoehe);
    s:="Zeichensatz AbsH: "; s1:=CardToStr(AbsHoehe,2);
    s2:=CardToStr(ChWeite,2); s3:=CardToStr(ChHoehe,2);
    Append(s1,s,ok); Append(', ',s,ok); Append(s3,s,ok);
    Append(' * ',s,ok); Append(s2,s,ok);
    INC(y,AbsHoehe+3);
    GrafText(Geraet,Pnt(10,y),s);
    INC(AbsHoehe);
    IF AbsHoehe=12 THEN INC(AbsHoehe) END  (* Abshoehe 12 führt bei ...*)
  UNTIL AbsHoehe = 26;                     (* Megamax-Mod. zum Absturz *)
  REPEAT UNTIL KeyPressed();
  abmelden
END GrafikZeichenSaetze.
```

4.6.4 Ein externer Modul für VDI-Grafik

Bei den folgenden Beispielen geht es um grafische Darstellung von Szenarien, die auf reellen Werten basieren (also nicht in handlichen Pixel-Koordinaten in INTEGER). Dabei tritt das Problem auf, daß die reellen Koordinaten, im folgenden »Weltkoordinaten« genannt, in

Pixelkoordinaten (»Bildkoordinaten«) umgerechnet werden müssen. Dieses Problem trat schon bei dem Mandelbrot-Programm auf. Diese Rechnerei versteckt man am besten in einem externen Modul »`GrafikWelt`«. Dadurch werden nun die folgenden Programme sehr handlich.

Dieser Modul baut auf unserem Modul `Grafik` auf und enthält neben den Routinen zur Konvertierung der Weltkoordinaten in Bildkoordinaten einige Prozeduren zum direkten Zeichnen mit Weltkoordinaten. Zu erwähnen ist auch die leistungsfähige Routine `AchsenKreuz`, die eine vollständige Skalierung und Beschriftung eines Achsenkreuzes übernimmt. Graphen sehen damit gleich erheblich besser aus.

Ein Anwenderprogramm kann wie gewohnt `SetzeBereich` aus `Grafik` aufrufen. Dann teilt es dem Modul `GrafikWelt` über `SetzeSkalierung` den Bereich der Weltkoordinaten mit, die es auf den Arbeitsbereich abgebildet haben will:

```
MODULE EinGrafikModul;
IMPORT Grafik, GrafikWelt;
<...>
BEGIN    (* ----    Hauptprogramm    ----- *)
  Grafik.anmelden;
  Grafik.Hintergrund;
  Grafik.SetzeBereich(<....>);          (* hier Pixelkoordinaten *)
  GrafikWelt.SetzeSkalierung(<....>);   (* hier Weltkoordinalten *)
  GrafikWelt.Achsenkreuz(<....>);
  <Anwender-Anweisungen>
  Grafik.abmelden
END EinGrafikModul.
```

```
DEFINITION MODULE GrafikWelt;
        (*
         *  Dient zur Bearbeitung von Grafiken in Koordinatensystemen.
         *  Sämtliche zu übergebende Koordinaten sind "Weltkoordinaten",
         *  d.h. sie entsprechen realen Größen, die von den Prozeduren
         *  dieses Moduls in Bildschirmkoordinaten umgerechnet werden.
         *  Vor der Benutzung ist "anmelden" und "SetzeBereich" aus dem
         *  Modul "Grafik" aufzurufen, am Ende "abmelden".            *)

FROM GrafBase IMPORT Point;

        (* =========== Initialisierungsprozedur ============ *)

PROCEDURE SetzeSkalierung(x1,y1, x2, y2: REAL);
        (*
         *  Berechnet die Skalierungsfaktoren für die Grafik.
```

```
        *  Dabei ist (xl,x2) die äußerste Weltkoordinate links unten
        *  und (x2,y2) rechts oben.
        *)

        (* ============= Maßstabskonvertierung ============= *)

PROCEDURE KonvertX(xW:REAL) : INTEGER;
        (*
        * Konvertiert die x-Weltkoordinate in die x-Bildschirmkoordinate.
        *)

PROCEDURE KonvertY(yW:REAL) : INTEGER;
        (*
        * Konvertiert die y-Weltkoordinate in die y-Bildschirmkoordinate.
        *)

PROCEDURE KonvertP(xW,yW : REAL) : Point;
        (*
        * Konvertiert den Weltpunkt (xW,yW) in den entsprechenden Bildpunkt.
        *)

PROCEDURE KonvertLx(xAbst : REAL) : INTEGER;
        (*
        * Konvertiert den x-Abstand  der "Welt" in den des Bildschirms.
        *)

PROCEDURE KonvertLy(yAbst : REAL) : INTEGER;
        (*
        * Konvertiert den y-Abstand der "Welt" in den des Bildschirms.
        *)

        (* =============== Zeichen Prozeduren =============== *)

PROCEDURE PlotW(xW,yW: REAL);
        (*
        * Zeichnet den dem Weltpunkt (xW,yW) entsprechenden Bildpunkt.
        *)

PROCEDURE LineW(xl,yl, x2,y2: REAL);
        (*
        *  Zeichnet eine Linie von (xl,yl) nach (x2,y2) (Weltkoordinaten).
        *)

        (* ================= Verschiedenes  ================= *)
```

```
PROCEDURE HoleMausW(VAR xW,yW : REAL);
        (*
         *  Diese Prozedur wartet auf das Klicken der
         *  linken Maustaste. Sie übergibt dann die
         *  dem Mauscursor entsprechenden Bildschirm-
         *  koordinaten umgerechnet in Weltkoordinaten xW,yW.
         *)

PROCEDURE AchsenKreuz(xEinheit, yEinheit : REAL;
                      xDezimalen,  yDezimalen  : CARDINAL;
                      xAchsenText, yAchsenText : ARRAY OF CHAR);
        (*
         *  Zeichnet ein Achsenkreuz mit der Skala "xEinheit"
         *  auf der x-Achse in Weltkoordinaten, entsprechend
         *  "y-Einheit". Die Beschriftung der Skalen erfolgt
         *  mit dezimalen Nachkommastellen. Die beiden Achsen
         *  werden mit den Texten "xAchsenText" bzw.
         *  "yAchsenText" versehen. Die Routine
         *  prüft selbständig, wo genug Platz für die Beschrif-
         *  tungen ist. Sind die ersten Parameter = 0.0, so
         *  wird die Skala unterdrückt. Die Beschriftung wird
         *  mit leeren Strings unterdrückt.
         *)

END GrafikWelt.
```

Die Implementation der Routine `Achsenkreuz` sieht relativ aufwendig aus. Sie muß aber automatisch viele ungünstige Fälle berücksichtigen, denn auch wenn die Achsen nahe am Bildschirmrand liegen, muß ein geeigneter Platz für die Beschriftung gefunden werden. Anfangs sollten Sie die Implementation ohnehin übergehen und gleich zu den Anwendungen weiterblättern.

```
IMPLEMENTATION MODULE GrafikWelt;

FROM MathLib0       IMPORT entier;
FROM Strings        IMPORT Length,String;
FROM StrConv        IMPORT RealToStr;
FROM GrafBase       IMPORT Point, Pnt;
FROM GEMGlobals     IMPORT MarkerType, LineType;
FROM VDIOutputs     IMPORT FillRectangle, Line, Mark;
FROM VDIAttributes  IMPORT SetMarkerType, SetLineType, SetLineColor;
FROM VDIInputs      IMPORT HideCursor, ShowCursor;
FROM Grafik         IMPORT Geraet, ArbeitsBereich, HoleMaus, Schreibe;
```

```
VAR
   GrafParm : RECORD
                 xBMin, yBMin, xBMax, yBMax : INTEGER;  (* Bildschirmausmaße  *)
                 xWMin, yWMin, xWMax, yWMax : REAL;     (* "Welt"- Außmaße    *)
                 masstabX, masstabY         : REAL      (* Umrechnungsfaktoren*)
              END;

PROCEDURE SetzeSkalierung(x1,y1, x2, y2: REAL);
BEGIN
  WITH GrafParm DO
    WITH ArbeitsBereich DO
      xBMin:= xMin; yBMin:=yMin; xBMax:=xMax; yBMax:=yMax
    END;
    xWMin:=x1; yWMin := y1; xWMax := x2; yWMax := y2;
    masstabX := FLOAT(xBMax - xBMin + 1) / (xWMax - xWMin);
    masstabY := FLOAT(yBMax - yBMin + 1) / (yWMax - yWMin);
  END;
  (* ---- Eine Voreinstellung.... (paßt hier ganz gut) ---- *)
  SetMarkerType(Geraet, pointMark);                          (* für 'PlotW' *)
END SetzeSkalierung;

PROCEDURE KonvertX(xW:REAL) : INTEGER;
BEGIN
  WITH GrafParm DO
    RETURN xBMin + SHORT(entier(masstabX * (xW-xWMin)))
  END
END KonvertX;

PROCEDURE KonvertY(yW:REAL) : INTEGER;
BEGIN
  WITH GrafParm DO
    RETURN yBMax - SHORT(entier(masstabY * (yW-yWMin)))
  END
END KonvertY;

PROCEDURE KonvertP(xW,yW : REAL) : Point;
BEGIN
  RETURN Pnt(KonvertX(xW), KonvertY(yW))
END KonvertP;

PROCEDURE KonvertLx(xAbst : REAL) : INTEGER;
BEGIN
  RETURN SHORT(entier(GrafParm.masstabX * xAbst))
END KonvertLx;
```

```
PROCEDURE KonvertLy(yAbst : REAL) : INTEGER;
BEGIN
  RETURN SHORT(entier(GrafParm.masstabY * yAbst))
END KonvertLy;

PROCEDURE PlotW(xW,yW: REAL);
BEGIN
  Mark(Geraet, KonvertP(xW,yW))
END PlotW;

PROCEDURE LineW(x1,y1, x2,y2: REAL);
BEGIN
  Line(Geraet, KonvertP(x1,y1), KonvertP(x2,y2));
END LineW;

PROCEDURE HoleMausW(VAR xW,yW : REAL);
VAR
    x,y: INTEGER;
BEGIN
    HoleMaus(x,y);
    WITH GrafParm DO
      xW := xWMin + FLOAT(x-xBMin) / masstabX;
      yW := yWMin + FLOAT(yBMax-y) / masstabY
    END
END HoleMausW;

PROCEDURE AchsenKreuz(xEinheit,      yEinheit     : REAL;
                      xDezimalen ,   yDezimalen   : CARDINAL;
                      xAchsenText,   yAchsenText  : ARRAY OF CHAR);
CONST
    dsx = 4; dsy = 4;                          (* Strichlängen für die Skalen *)
VAR
    OrgX, OrgY  : INTEGER;                    (* Nullpunkt in Bildkoordinaten *)
    justX, justY : INTEGER;                   (* Zur Textjustierung          *)
    e           : REAL;
    s           : String;

  PROCEDURE XAchseSkalieren;
  BEGIN
    WITH GrafParm DO
      justY := OrgY + 7 + dsy;                        (* Zahlen unter die Achse *)
      IF justY > yBMax THEN justY := OrgY - dsy -1 END; (* unten kein Platz *)
      e := xEinheit;                                  (* pos. x-Achse skalieren *)
```

```
    WHILE e < xWMax DO
      Line(Geraet, Pnt(KonvertX(e),OrgY+dsy), Pnt(KonvertX(e),OrgY-dsy));
      s := RealToStr(e, 0, xDezimalen);
      Schreibe(KonvertX(e)-INTEGER((Length(s) DIV 2))*6, justY,1, s);
      e := e + xEinheit
    END;
    e := - xEinheit;                          (* neg. x-Achse skalieren *)
    WHILE e > xWMin DO
      Line(Geraet, Pnt(KonvertX(e),OrgY+dsy), Pnt(KonvertX(e),OrgY-dsy));
      s := RealToStr(e,0,xDezimalen);
      Schreibe(KonvertX(e)-INTEGER((Length(s) DIV 2))*6, justY,1, s);
      e := e - xEinheit
    END
  END
END XAchseSkalieren;

PROCEDURE YAchseSkalieren;
VAR laenge : CARDINAL;          (* Maximallänge für die Skalenbeschriftung *)
BEGIN
  WITH GrafParm DO
    e := yEinheit;      (* größte Zahlenlänge auf der pos. Achse ermittelen *)
    laenge:=0;
    WHILE e < yWMax DO
      s:= RealToStr(e,0,yDezimalen);
      IF laenge < Length(s) THEN laenge:=Length(s) END;
      e:=e + yEinheit;
    END;
    e := -yEinheit;             (* gibt es auf der neg. Achse größere Werte ? *)
    WHILE e > yWMin DO
      s:=RealToStr(e,0,yDezimalen);
      IF laenge < Length(s) THEN laenge:=Length(s) END;
      e := e - yEinheit;
    END;
    justX := OrgX - INTEGER(laenge)*6-dsx-2;  (* Zahlen links an d. Achse *)
    IF justX < xBMin THEN justX:= OrgX + dsx +2 END; (*  links kein Platz *)
    e := yEinheit;                            (* pos. y-Achse skalieren *)
    WHILE e < yWMax DO
      Line(Geraet, Pnt(OrgX+dsx,KonvertY(e)), Pnt(OrgX- dsx,KonvertY(e)));
      s := RealToStr(e,0,yDezimalen);
      Schreibe(justX, KonvertY(e),1, s);
      e := e + yEinheit
    END;
    e := -yEinheit;                              (* neg. Achse skalieren *)
    WHILE e > yWMin DO
```

```
        Line(Geraet, Pnt(OrgX+dsx,KonvertY(e)), Pnt(OrgX- dsx,KonvertY(e)));
        s := RealToStr(e,0,yDezimalen);
        Schreibe(justX, KonvertY(e),1, s);
        e := e - yEinheit
      END
    END
  END YAchseSkalieren;

  PROCEDURE XAchseBeschriften;
  BEGIN
    WITH GrafParm DO
      justY := OrgY + 21 + dsy;                    (* Text unter die Achse *)
      IF justY > yBMax THEN justY := OrgY - dsy - 8 END;(* unten kein Platz *)
      justX := xBMax - 8*INTEGER(Length(xAchsenText)) - 2*dsx;
      Schreibe(justX, justY, 3, xAchsenText)
    END
  END XAchseBeschriften;

  PROCEDURE YAchseBeschriften;
  BEGIN
    WITH GrafParm DO
      justX := OrgX - 8*INTEGER(Length(yAchsenText)) - dsx;    (* Text links *)
      IF justX < xBMin THEN justX := OrgX + dsx END;        (* links kein Platz *)
      Schreibe(justX, yBMin+20,3,yAchsenText)
    END
  END YAchseBeschriften;

BEGIN
  SetLineType(Geraet,solidLn);
  HideCursor(Geraet);
  WITH GrafParm DO
    OrgX := KonvertX(0.0);                           (* Nullpunkt ermitteln *)
    OrgY := KonvertY(0.0);
    IF (OrgX < xBMin) OR (xBMax < OrgX) THEN OrgX := xBMin END;
    IF (OrgY < yBMin) OR (yBMax < OrgY) THEN OrgY := yBMax END;
    IF (xWMin <= 0.0) & (0.0 <= xWMax) THEN
       Line(Geraet, Pnt(xBMin,OrgY), Pnt(xBMax,OrgY));    (* x-Achsezeichnen *)
      IF xEinheit > 0.0 THEN  XAchseSkalieren END;
      XAchseBeschriften
    END;
    IF (yWMin <= 0.0) AND (0.0 <= yWMax) THEN
       Line(Geraet, Pnt(OrgX,yBMin), Pnt(OrgX,yBMax));    (* y-Achsezeichnen *)
      IF yEinheit > 0.0 THEN  YAchseSkalieren END;
      YAchseBeschriften
```

```
      END;
    END;
    ShowCursor(Geraet,FALSE);
  END AchsenKreuz;

  END GrafikWelt.
```

Die beiden nächsten Beispiele sind etwas komplexer: Es geht um Simulationen aus der Biologie und der Physik. Wir stellen jeweils die mathematischen Grundlagen zu den Programmen dar. Wenn Sie keine Mathematik mögen, überschlagen Sie diese Abschnitte einfach. Sie sind für das Verständnis der weiteren Abschnitte unwesentlich.

4.6.5 Der Kampf ums Dasein

Auf einer Insel mit üppiger Vegetation leben Füchse und Kaninchen. Die Kaninchen ernähren sich von dem Gras und die Füchse von den Kaninchen. Nun passiert folgendes: Wenn zu viele Füchse da sind, werden die Kaninchen rasch dezimiert. Das führt dazu, daß die Füchse kaum noch Nahrung finden, und sie fangen an, selbst einzugehen. Durch das Verschwinden der Füchse können sich aber die paar Kaninchen, die noch übrig sind, wieder ungestört vermehren. Und auf einmal ist dann wieder Nahrung für die Füchse da...

Im mathematischem Modell sieht das folgendermaßen aus: Zu einer Zeit `t` leben `K(t)` Kaninchen auf der Insel. Wenn wir die Füchse erst einmal weglassen, würden sich die Kaninchen innerhalb eines Zeitintervalls Δt vermehren. Der Zuwachs der Kaninchen ΔK wird proportional zu ihrer Anzahl und der Länge des Zeitintervalls Δt sein, also

$\Delta K \sim K*\Delta t$, daraus folgt:

$\Delta K = a*K*\Delta t$

wobei `a` Proportionalitätsfaktor ist (etwa »Geburtenrate«). Hiermit folgt:

$$(I)\ K(t+\Delta t) = K(t) + a*K(t)*\Delta t$$

Betrachten wir nun die Füchse. Gäbe es keine Kaninchen, so wird deren Anzahl `F(t)` abnehmen, da sie allmählich verhungern müßten:

$$(II)\ F(t+\Delta t) = F(t) - c*F(t)*\Delta t$$

Den Proportionalitätsfaktor `c` könnte man mit »Sterberate« bezeichnen. Nun leben aber Füchse und Kaninchen zusammen und das nicht gerade friedlich. Jede Begegnung von Füch-

sen und Kaninchen dezimiert die Kaninchen (werden gefressen), läßt aber die Zahl der Füchse steigen (weil sie dadurch satt werden). Die Anzahl der Begegnungen sind aber proportional zur Anzahl der Kaninchen `K(t)`, zur Anzahl der Füchse `F(t)` und natürlich zur Länge des betrachteten Zeitintervalls `Δt`. Wir modifizieren also `(I)` und `(II)`:

```
(I') K(t + Δt) = K(t) + a*K(t)*Δt - b*K(t)*F(t) *Δt
               = K(t)*(1 + (a - b*F(t))*Δt)
(II')F(t + Δt) = F(t) - c*F(t)*Δt + d*K(t)*F(t)*Δt
               = F(t)*(1 - (c - d*K(t))*Δt)
```

Erstaunlich ist nun, daß sich die Kaninchen- und Fuchspopulation aufgrund von fressen und gefressen werden periodisch entwickeln. Das Ganze ist keine Spielerei, sondern man entdeckte die Periodizität in der Natur und konnte sie zunächst nicht erklären. Es wurden sogar (mal wieder) die Sonnenflecken dafür verantwortlich gemacht. Aber dahinter steckt mathematisch eine sogenannte »gekoppelte Differentialgleichung«, die nach den Mathematikern *Voltera* und *Lotka* benannt ist. Das folgende Programm gibt Aufschluß. Es zeigt die Kurven `K(t)` und `F(t)` für die Anfangswerte `K(0)=30000` und `F(0)=1500`. Die Konstanten a, b, c, d wurden empirisch ermittelt. Die Rechnung ergibt eine Periodizität von 3,7 Jahren, in der Natur sind es 4 Jahre, da durch den Winter einige Unregelmäßigkeiten mit ins Spiel kommen.

```
MODULE KampfUmsDaseinl;

FROM Terminal    IMPORT KeyPressed;
FROM MathLib0    IMPORT entier;
FROM VDIInputs   IMPORT HideCursor, ShowCursor;
FROM Grafik      IMPORT anmelden, abmelden, Geraet,
                        Hintergrund, SetzeBereich ,Schreibe;
FROM GrafikWelt  IMPORT SetzeSkalierung, PlotW, AchsenKreuz;
FROM GEMGlobals  IMPORT GemChar;
FROM AESEvents   IMPORT KeyboardEvent;

CONST
      a = 3.86;                 (* a,b,c,d empirisch gefundene ... *)
      b = 0.00177;             (* ...Konstanten für die Population *)
      c = 0.823;
      d = 0.0000338;
      dt = 0.001;
      MaxJahre = 10.0;

VAR
    K,Kl,F,t: REAL;
```

```
PROCEDURE rechnen;
BEGIN
  t := 0.0;
  HideCursor(Geraet);              (* Maus beim Zeichnen wegmachen... *)
  REPEAT
    Kl:= K;
    K := K + ( a * K - b * K  * F) * dt;
    F := F + (-c * F + d * Kl * F) * dt;
    t := t + dt;
    PlotW(t, K);
    PlotW(t, F);
  UNTIL (MaxJahre < t) OR KeyPressed();
  ShowCursor(Geraet,FALSE)        (* Maus zeigen (muß ja da sein...) *)
END rechnen;

VAR taste: GemChar;

BEGIN
  anmelden;
  Hintergrund;
  SetzeBereich(9,9, 630,390);
  SetzeSkalierung(-1.0, -5000.0, MaxJahre, 60000.0);
  AchsenKreuz(1.0, 10000.0, 1,0, "t/Jahre", "K, F");
  Schreibe(150,50,4,"Fuchs-Kaninchen-Problem");
  K := 30000.0;
  F := 1500.0;
  rechnen;
  KeyboardEvent(taste);
  abmelden
END KampfUmsDaseinl.
```

Noch überraschender als die Darstellung der Kurven `K(t)` und `F(t)` über der Zeit `t` ist die Darstellung von `F` über `K`. Ihr liegt die Fragestellung zu Grunde: Wieviele Füchse gibt es bei einer bestimmten Kaninchenzahl? Man erhält eine geschlossene Kurve. Damit man verschiedene Anfangswerte `K(0)` und `F(0)` bequem ausprobieren kann, setzen wir die Maus ein: Beliebige Werte `K` und `F` werden durch Anklicken auf dem Bildschirm an der entsprechenden Position eingegeben. Intern werden die Werte durch die Prozedur `Anklicken` von der Maus gelesen und direkt in die richtigen Werte (»Weltkoordinaten«) umgerechnet. Wenn man mehrere verschiedene Punkte für die Kaninchen und Füchse angibt, erhält man verschiedene geschlossene Kurven, die einen gemeinsamen Mittelpunkt haben, den Gleichgewichtspunkt (er liegt bei `K = 24350` und `F = 2180` . Das kann man direkt den Gleichungen `(I')` und `(II')` entnehmen; die Klammerausdrücke müssen dabei `1` ergeben).

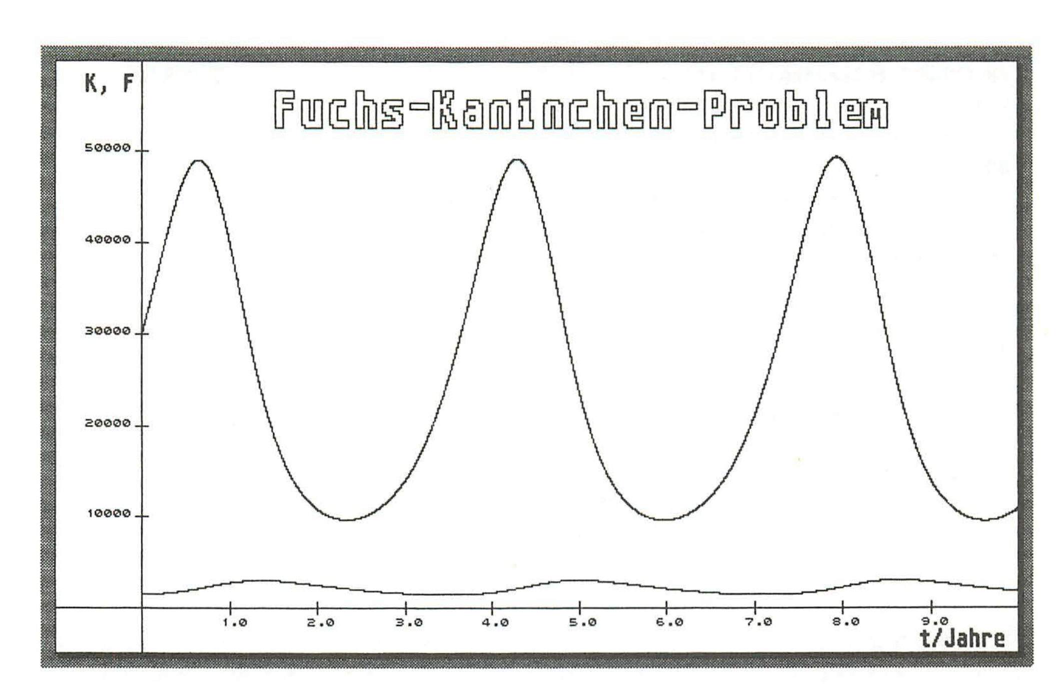


Bild 4.15: Das Fuchs-Kaninchen-Problem

```
MODULE KampfUmsDasein2;

FROM Terminal    IMPORT BusyRead;
FROM AESForms    IMPORT FormAlert;
FROM VDIInputs   IMPORT ShowCursor, HideCursor;
FROM Grafik      IMPORT anmelden, abmelden, Geraet, Hintergrund, SetzeBereich;
FROM GrafikWelt  IMPORT SetzeSkalierung, PlotW, AchsenKreuz, HoleMausW;

CONST a = 3.86;              (* Bedeutung der Werte s. KampfUmsDasein1 *)
      b = 0.00177;
      c = 0.823;
      d = 0.0000338;
     dt = 0.001;
     xH = 639;
     yH = 399;

VAR
     K,K1,F,t: REAL;
     ch      : CHAR;
```

```
PROCEDURE BedienungsAnleitung;
VAR s          : ARRAY [0..80] OF CHAR;
    AntwortKnopf : CARDINAL;
BEGIN
  s:="[3][Mausklick setzt Anfangswerte| |Taste unterbricht| |ESC = Ende][Ok]";
  FormAlert(1,s,AntwortKnopf)
END BedienungsAnleitung;

PROCEDURE rechnen;
BEGIN
  t := 0.0;
  HideCursor(Geraet);
  REPEAT
    Kl := K;
    K := K + ( a * K - b * K  * F) * dt;
    F := F + (-c * F + d * Kl * F) * dt;
    t := t + dt;

    PlotW(K,F);
    BusyRead(ch);
  UNTIL ch # 0C;
  ShowCursor(Geraet,FALSE);
END rechnen;

BEGIN
  anmelden;
  Hintergrund;
  BedienungsAnleitung;
  SetzeBereich(10,10, xH-10,yH-10);
  SetzeSkalierung(-7000.0,-400.0, 70000.0, 4000.0);
  AchsenKreuz(10000.0, 500.0,0,0, "Kaninchen", "Füchse");
  REPEAT
    HoleMausW(K,F);
    rechnen;
  UNTIL ch = 33C;               (* ESC: Fertig *)
  abmelden
END KampfUmsDasein2.
```

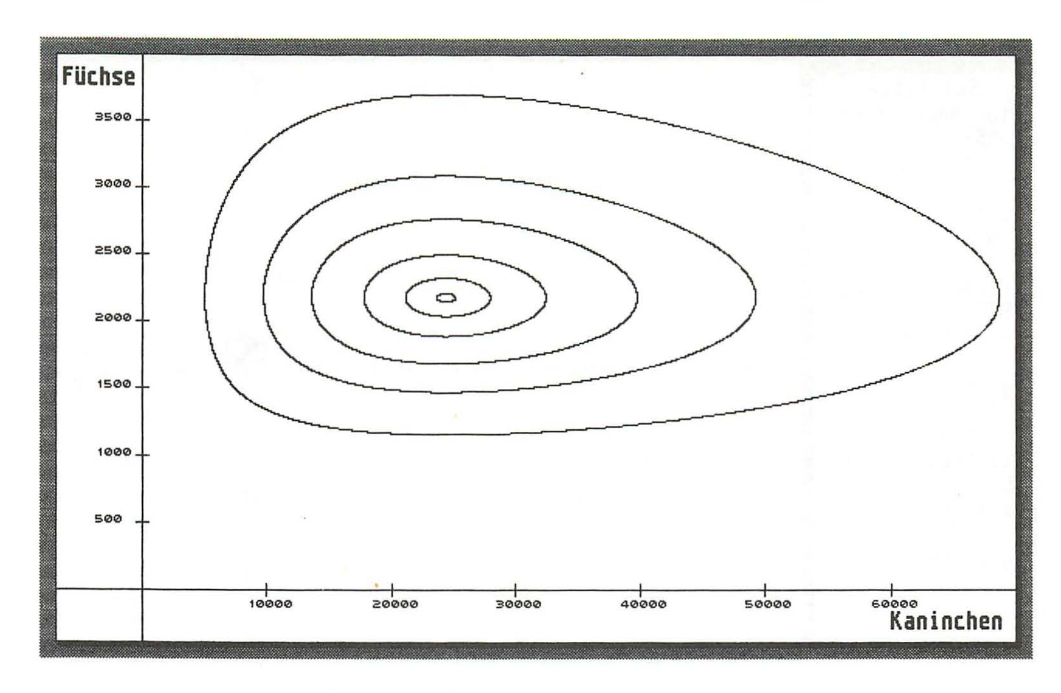


Bild 4.16: Darstellung der Fuchszahl über der Kaninchenzahl

4.6.6 Kepler, Newton und Atari

Das zweite Beispiel ist von den Berechnungen her noch etwas interessanter. Es stellt die Bahnen von Satelliten dar, die zum Beispiel in 35875 km über dem Äquator tangential von einer Rakete mit den Anfangsgeschwindigkeiten 3.071 km/s, 3.8 km/s und 4.8 km/s abgeschossen wurden. Der erste Wert entspricht der Geschwindigkeit eines »Synchronsatelliten« der sich auf einer Kreisbahn mit der Umlaufszeit 24 Stunden bewegt, also scheinbar fest über einem Ort der Erde steht (Fernsehsatellit). Mit der zweiten Geschwindigkeit erhält man eine Ellipsenbahn, mit der dritten eine Hyperbelbahn.

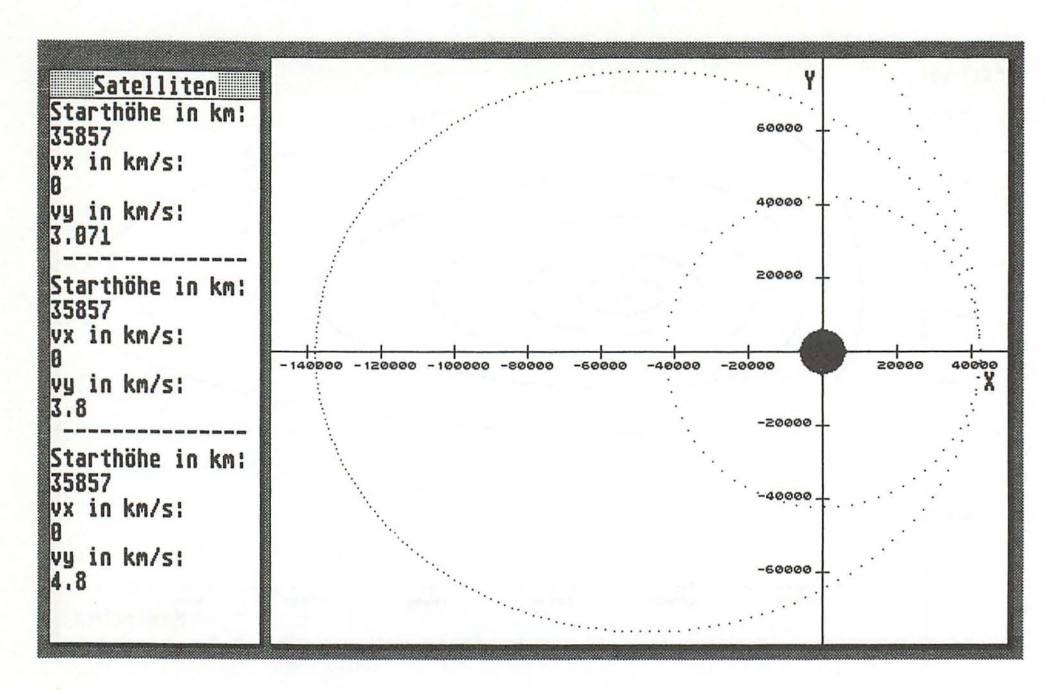


Bild 4.17: Drei Satellitenbahnen

Wir erläutern den Algorithmus für die Bahnkurve. Die Erklärungen sind hier etwas breiter, dafür lassen sie sich auf andere Beispiele übertragen. Auch diesen Abschnitt können Sie getrost übergehen, wenn Sie sich nicht mit den mathematischen und physikalischen Grundlagen herumschlagen möchten.

Da die Bewegung in der Ebene, die vom Erdmittelpunkt und der Kurve des Satellitenmittelpunktes aufgespannt wird, genügen zwei Koordinaten x und y zur Darstellung des Satelliten. Mittelpunkt des Koordinatensystems ist der Erdmittelpunkt.

Mit v_x, v_y seien die Komponenten des Geschwindigkeitsvektors $\vec{v}$ genannt, analog mit a_x, a_y die des Beschleunigungsvektors $\vec{a}$. Aus der Mechanik brauchen wir nun die Formeln:

(i) $\vec{F} = m \cdot \vec{a}$ (Kraft = Masse Beschleunigung)

(ii) $\vec{a} = \lim\limits_{\Delta t \to 0} \frac{\vec{v}(t + \Delta t) - \vec{v}(t)}{\Delta t}$ (Beschl. = Geschwindigkeitsänderungen pro Zeit Δt)

(iii) $\vec{v} = \lim\limits_{\Delta t \to 0} \frac{\vec{s}(t + \Delta t) - \vec{s}(t)}{\Delta t}$ (Geschwindigkeit = Wegänderung pro Zeit Δt)

Die Gleichungen (i)–(iii) beschreiben nun ein Prinzip, daß nicht nur bei diesem Beispiel gilt. Vielmehr dient es allgemein dazu, die Bahn eines Körpers zu berechnen: Aus der jeweils herrschenden Kraft errechnet man mittels (i) die Beschleunigung, aus dieser dann mit (ii) die neue Geschwindigkeit nach der Zeitspanne Δt und hieraus mit (iii) letztlich den neuen Ort. Hierzu muß lediglich noch der anfängliche Ort s(0) und die Anfangsgeschwindigkeit v(0) bekannt sein. Natürlich können wir nicht die Grenzwertbildung von (ii) und (iii) mit dem Rechner durchführen. Ist aber das betrachtete Zeitintervall hinreichend kurz, so folgt aus (i) und (ii) näherungsweise:

(ii') $\vec{v}(t+\Delta t) = \vec{v}(t) + \vec{a} \cdot \Delta t$
(iii') $\vec{s}(t+\Delta t) = \vec{s}(t) + \vec{v} \cdot \Delta t$

d. h. die Geschwindigkeit und der Ort läßt sich am Intervallende durch die Kenntnis ihrer Werte am Inervallanfang bestimmen. Man kann also »in die Zukunft sehen«!
Schreiben wir (ii') und (iii') noch komponentenweise, so folgt

(ii') $v_x(t+\Delta t) = v_x(t) + a_x \cdot \Delta t$ und $v_y(t+\Delta t) = v_y(t) + a_y \cdot \Delta t$

(iii') $s_x(t+\Delta t) = s_x(t) + v_x \cdot \Delta t$ und $s_y(t+\Delta t) = s_y(t) + v_y \cdot \Delta t$

Nun fehlt nur noch die computergerechte Aufbereitung der Formel (i). Wir stellen hier nach $\vec{a}$ um und erhalten komponentenweise geschrieben:

(i') $a_x = \frac{1}{m} \cdot F_x$ und $a_y = \frac{1}{m} \cdot F_y$

Fehlt nur noch das Kraftgesetz! Newton hat es herausgefunden, als er unter einem Apfelbaum lag und herunterfallende Früchte betrachtete:

(iv) $F = -f \frac{m_E \cdot m}{r^2}$

Dabei ist m_E die Erdmasse, m die des Satelliten, r der Abstand Erdmittelpunkt zu Satellit und f eine Konstante, die sogenannte Gravitationskonstante. Das Minuszeichen rührt daher, daß es sich um eine anziehende Kraft handelt (vgl. Abb).

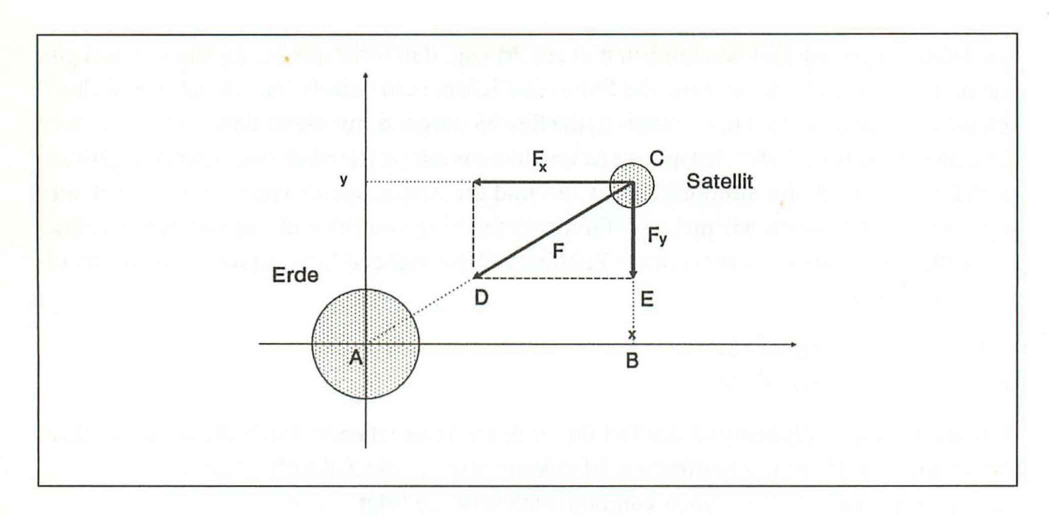


Bild 4.18: Gravitationskraft zwischen Erde und Satellit

Aus der Ähnlichkeit der Dreiecke ABC und DEC ergibt sich nun für die Kraftkomponenten F_x und F_y:

(vi) $F_x = -\frac{F}{r}\cdot x$ und $F_y = -\frac{F}{r}\cdot y$

also folgt insgesamt:

(i") $a_x = -fm_E \frac{x}{r^3}$ und $a_y = -fm_E \frac{y}{r^3}$

Den Abstand r erhält man nach dem Satz des Pythagoras zu $r = \sqrt{x^2 + y^2}$.

Programmiert man nun die Gleichungen (iii'), (i") und (ii") hintereinander, so erhält man die gesuchten Bahnkurven, die schon Kepler beschrieb.

Es ergibt sich jedoch eine größere Abweichung von der tatsächlichen Satellitenbahn (von der Erde weg), da die Beschleunigung und hieraus die neue Geschwindigkeit stets mit den Werten vom Intervallanfang ermittelt werden. Während der Zeitspanne Δt ändern sich aber diese Werte. Eine Möglichkeit ist es, Δt sehr klein zu wählen (zum Beispiel eine Sekunde), was aber die Rechenzeit erhöht. Besser geht es mit folgendem Trick, dem »Halbschrittverfahren«:

Einmal zu Beginn ermittelt man die Geschwindigkeit in der Intervallmitte (also zur Zeit $\Delta t/2$), errechnet daraus den Ort und die Beschleunigung zur Zeit Δt, dann daraus wiederum die Geschwindigkeit bei $\Delta t + \Delta t/2$ usw. Die einmalige Vorwegberechnung ist unaufwendig, erhöht die Genauigkeit aber erheblich, so daß wir mit einer Zeitspanne von $\Delta t = 15$ Sekunden gute Werte erhalten.

```
MODULE SatellitenBahnen;

FROM GrafBase      IMPORT black;
FROM GEMGlobals    IMPORT MarkerType;
FROM VDIOutputs    IMPORT Circle;
FROM VDIAttributes IMPORT SetFillColor;
FROM VDIInputs     IMPORT HideCursor, ShowCursor;
FROM AESForms      IMPORT FormAlert;
FROM TextWindows   IMPORT Window,WindowQuality,ForceMode, KeyPressed,
                          Open, Close, WQualitySet, ShowMode,
                          ReadString, WriteString, WriteLn, WritePg;
FROM MathLib0      IMPORT sqrt, entier;
FROM StrConv       IMPORT StrToReal;
FROM Sound         IMPORT Einstellen, Ton, Aufhoeren;
FROM Grafik        IMPORT anmelden, abmelden, Geraet, Hintergrund, SetzeBereich;
FROM GrafikWelt    IMPORT SetzeSkalierung, PlotW,
                          AchsenKreuz, KonvertP, KonvertLx;

 CONST
      dt  = 15.0;          (* Sekunden-Abstand zwischen 2 Berechnungen  *)
      sec = 1200.0;        (* Abstand zwischen 2 Plots (= 20 Minuten)   *)
      me  = 5.974E24;      (* Erdmasse in kg *)
      re  = 6370.3;        (* mittlerer Erdradius in km *)
      f   = 6.67E-20;      (* Gravitationskonstante in km^3/(kg s^2)    *)
      k   = -me*f*dt;      (* Konstante für die Beschleunigungen ax, ay *)

VAR
    x, y, vx, vy, ax, ay : REAL;  (* Ort-, Geschw.-, Beschl. Koo. des Sat.*)
    radius               : REAL;  (* Abstand Satellit - Erdmittelpunkt    *)
    hoehe                : REAL;  (* Anfangshöhe über der Erdoberfläche   *)
    t                    : REAL;  (* Zeit, wird zw. 2 Plots hochgezählt   *)
    EingabeFenster       : Window;
    ok, abgestuertzt     : BOOLEAN;

PROCEDURE InitBildschirm;
VAR  einheit,i : INTEGER;
BEGIN
  SetzeBereich(150,10,630,390);
  SetzeSkalierung(-150000.0, -80000.0, 50000.0, 80000.0);
  AchsenKreuz(20000.0, 20000.0,0,0, "X", "Y");
  HideCursor(Geraet);
  SetFillColor(Geraet,black);
  Circle(Geraet,KonvertP(0.0, 0.0),KonvertLx(re));         (* Erde zeichnen *)
  ShowCursor(Geraet,FALSE)
```

```
END InitBildschirm;

PROCEDURE Eingabe;

  PROCEDURE LiesReal(s : ARRAY OF CHAR) : REAL;
  VAR pos  : CARDINAL;
      hilf : ARRAY[0..8] OF CHAR;
      ok   : BOOLEAN;
  BEGIN
    WriteLn(EingabeFenster);
    WriteString(EingabeFenster,s);
    WriteLn(EingabeFenster);
    ReadString(EingabeFenster,hilf);
    pos:=0;
    RETURN StrToReal(hilf,pos,ok)
  END LiesReal;

BEGIN
  WriteLn(EingabeFenster);
  WriteString(EingabeFenster," ---------------");
  hoehe:=LiesReal("Starthöhe in km:");
  vx:=LiesReal("vx in km/s:");
  vy:=LiesReal("vy in km/s:")
END Eingabe;

PROCEDURE SatellitRechnen;
VAR
  hilf, radHoch2, radHoch3 : REAL;
BEGIN
  x:=re+hoehe; y:=0.0; t:=0.0;
  PlotW(x,y);
  (* --------------------- Vorweg-Halbschritt ------------------------- *)
  radHoch2:=x*x+y*y;
  radius:=sqrt(radHoch2);
  radHoch3:=radHoch2*radius;
  hilf :=k/radHoch3/2.0;          (* für die Beschl. in der Intervallmitte *)
  vx:=vx+x*hilf;
  vy:=vy+y*hilf;
  (* ----------------------- Rechen-Schleife -------------------------- *)
  HideCursor(Geraet);
  REPEAT
    t:=t+dt;
    x:=x+vx*dt;
    y:=y+vy*dt;
```

```
    radHoch2:=x*x+y*y;
    radius:=sqrt(radHoch2);
    radHoch3:=radHoch2*radius;
    hilf:=k/radHoch3;
    vx:=vx+x*hilf;
    vy:=vy+y*hilf;
    IF t>=sec THEN PlotW(x,y); t:=t-sec END;
    abgestuertzt:=(radius < re);
    IF abgestuertzt THEN Einstellen(15); Ton(2400,50); Aufhoeren END;
  UNTIL abgestuertzt OR KeyPressed();
  ShowCursor(Geraet,FALSE);
END SatellitRechnen;

PROCEDURE fertig : BOOLEAN;
VAR knopf : CARDINAL;
    s     : ARRAY [0..80] OF CHAR;
BEGIN
  ShowCursor(Geraet,FALSE);
  IF abgestuertzt THEN
    s:="[1][Der Satellit ist abgestützt| |Was nun...][Eingabe|Löschen|Ende]"
  ELSE
    s:="[3][Was nun...][Eingabe|Löschen|Ende]"
  END;
  FormAlert(1,s,knopf);
  IF knopf=2 THEN InitBildschirm; WritePg(EingabeFenster); END;
  HideCursor(Geraet);
  RETURN knopf=3
END fertig;

BEGIN
  anmelden;
  Hintergrund;
  InitBildschirm;
  Open(EingabeFenster,80,25,WQualitySet{titled},noHideWdw,forceLine,
       "Satelliten",1,1,17,24,ok);
  REPEAT
    Eingabe;
    SatellitRechnen;
  UNTIL fertig();
  Close(EingabeFenster);
  abmelden
END SatellitenBahnen.
```

Falls Sie Gefallen an solchen physikalischen Simulationen gefunden haben, hier noch einige Anregungen, bei denen Sie in den obigen Gleichungen nur die jeweilige Kraftfunktion abzuändern brauchen:

- Simulieren Sie die Bahn eines schnellen α-Teilchens, daß in die Nähe eines positiv geladenen Atomkerns gelangt. Man braucht dann nur die Konstanten und das Vorzeichen der Kraft im Programm zu ändern!
- Simulieren Sie die Bahn eines Geschosses in der Luft oder trivialer einer Kugel beim Kugelstoßen. Welches ist bei letzterer der optimale Abwurfswinkel (eben nicht 45° wie man es in der Schule lernt!).
- Simulieren Sie die Bahn eines elektrisch geladenen Teilchen in speziellen elektrischen und magnetischen Feldern. Hierzu muß auch die z-Koordinate betrachtet werden. Implementieren Sie eine 3-D-Darstellung!
- Simulieren Sie einen Ball im Schwerefeld der Erde (an der Erdoberfläche gilt für die Kraft einfach $F_y = -g$, $F_x = 0$, ($g = 9.81\ m/s^2$)). Trifft der Ball auf den Boden, so wird er nach dem Reflexionsgesetz reflektiert (einfach `vy=-vy` setzen). Dies könnte schon in einem Spiel brauchbar sein.

4.6.7 Auswertung von Meßreihen (lineare Regression)

Den Abschluß dieses Grafikabschnittes bildet ein Programm, daß jeder brauchen kann, der es mit Messungen (in der Schule, im Labor) zu tun hat. Wir gehen von folgender Problemstellung aus:

Gegeben sei ein Experiment, das n Wertepaare (x_k, y_k) von Meßdaten liefert z.B.:

x_k	0.0	0.2	0.4	0.5	0.8	1.0	1.5	1.8
y_k	7.0	8.2	9.0	10.2	13.4	14.0	15.0	18.2

Die Meßdaten seien so gestaltet, daß man einen lineareren Zusammenhang `y = mx+b` vermutet, der Graph müßte also eine Gerade ergeben. Da man wegen Meßfehlern nicht erwarten kann, daß alle Meßwertepaare exakt auf einer Geraden liegen, ergibt sich die Frage nach der optimalen Geraden mit der Steigung `m` und dem Achsenabschnitt `b`, durch die die »Punktwolke« bestmöglich angenähert wird. Diese Gerade heißt *Ausgleichsgerade*.

Dieses Problem wird nach Gauß mit der »Methode der kleinsten Fehlerquadrate« gelöst.

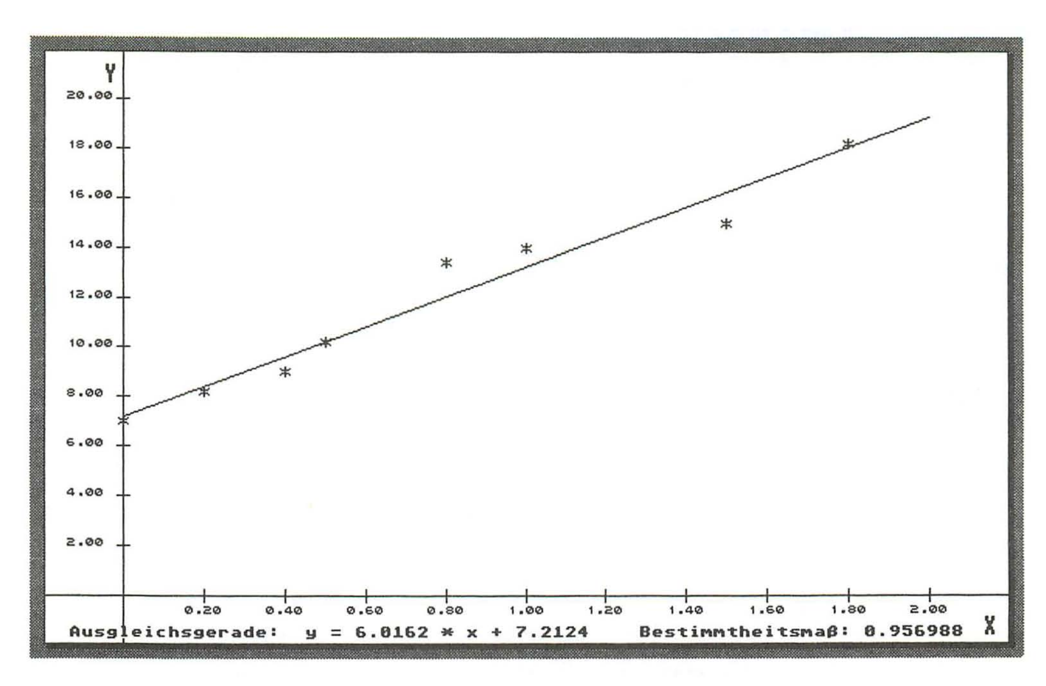


Bild 4.19: Ausgleichsgerade zu einer Meßreihe

Interessierte finden die zugehörige mathematische Herleitung im Anschluß an das Programm. Außer der Steigung und dem Achsenabschnitt der Ausgleichsgerade liefert das Programm ein Maß (»Bestimmtheitsmaß« r^2) dafür, wie »gut« die Meßpunkte auf einer Geraden liegen. Ist $r^2 = 1$, so bedeutet dies, daß alle Punkte genau auf der Geraden liegen; umgekehrt ist für $r^2 = 0$ auf einen nichtlinearen Zusammenhang zu schließen. Als »normaler« Wert für eine lineare Meßreihe darf $r^2 = 0,9$, als »guter« Wert $r^2 = 0,99$ gelten.

```
MODULE MessReihen;

FROM MathLib0        IMPORT log, pwrOfTen, real, entier;  (* für Pr. "Glaetten" *)
FROM Strings         IMPORT String, Append;
FROM StrConv         IMPORT RealToStr;
FROM VDIInputs       IMPORT HideCursor, ShowCursor;
FROM VDIAttributes   IMPORT SetMarkerType;
FROM GEMGlobals      IMPORT GemChar, MarkerType;
FROM AESEvents       IMPORT KeyboardEvent;
FROM Grafik          IMPORT anmelden, abmelden, Geraet,
                            Hintergrund, SetzeBereich ,Schreibe;
FROM GrafikWelt      IMPORT SetzeSkalierung, PlotW, LineW, AchsenKreuz;
```

```
TYPE  MessWert = RECORD
                    x, y :REAL;
                  END;

VAR
  Werte                       : ARRAY [0..7] OF MessWert;
  Steigung, yAbschnitt, BMass : REAL;
  xMax, yMax                  : REAL;
  ok                          : BOOLEAN;

PROCEDURE Eingabe;                                     (* Test-Prozedur *)
BEGIN
  Werte[0].x := 0.0; Werte[0].y := 7.0;  Werte[1].x := 0.2; Werte[1].y := 8.2;
  Werte[2].x := 0.4; Werte[2].y := 9.0;  Werte[3].x := 0.5; Werte[3].y :=10.2;
  Werte[4].x := 0.8; Werte[4].y :=13.4;  Werte[5].x := 1.0; Werte[5].y :=14.0;
  Werte[6].x := 1.5; Werte[6].y :=15.0;  Werte[7].x := 1.8; Werte[7].y :=18.2;
END Eingabe;

PROCEDURE Berechnen(VAR Messung                       : ARRAY OF MessWert;
                    VAR Anstieg, Abschnitt, BestimmtMass : REAL;
                    VAR xmax, ymax                    : REAL;
                    VAR Fehler                        : BOOLEAN);
VAR
    sx, sx2, sy, sy2, sxy, zaehler, nennerX, nennerY, anzahl : REAL;
    i                                                        : CARDINAL;

BEGIN
   Fehler := FALSE;
   IF HIGH(Messung) < 1 THEN Fehler:= TRUE; RETURN END; (* mind. 2 Meßwerte *)
   sx:=0.0; sy:=0.0; sx2:=0.0; sy2:=0.0; sxy:=0.0; xmax:=0.0; ymax:=0.0;
   FOR i:=0 TO HIGH(Messung) DO
     WITH Messung[i] DO
       Fehler := (x < 0.0) ; Fehler := (y < 0.0 );      (* nur pos. Messwerte *)
       IF Fehler THEN RETURN END;                       (* .. sind zugelassen *)
       IF x > xmax THEN xmax:=x END;
       IF y > ymax THEN ymax:=y END;
       sx := sx + x; sy := sy + y;  sxy:= sxy + x*y;
       sx2:= sx2 + x*x; sy2:= sy2 + y*y;
     END
   END;
   anzahl := FLOAT(HIGH(Messung) + 1);
   zaehler := sxy - sx*sy/ anzahl;
   nennerX := sx2 - sx*sx/ anzahl;
   nennerY := sy2 - sy*sy/ anzahl;
```

```
   IF nennerX*nennerY = 0.0 THEN Fehler:=TRUE; RETURN END; (* nur gleiche ..*)
   Anstieg := zaehler / nennerX;                            (* ... Meßwerte  *)
   Abschnitt := (sy-Anstieg*sx) / anzahl;
   BestimmtMass := zaehler * zaehler / ( nennerX*nennerY);
END Berechnen;

PROCEDURE Ausgabe(VAR Messung                       : ARRAY OF MessWert;
                  VAR Anstieg, Abschnitt, BestimmtMass : REAL;
                  VAR xmax, ymax                    : REAL;
                  Fehler                            : BOOLEAN);
VAR
   i             : CARDINAL;
   s,s1, s2, s3  : String;
   ok, Minus     : BOOLEAN;
   taste         : GemChar;

  PROCEDURE Glaetten (VAR zahl : REAL);       (* Liefert glatte Werte für ... *)
  VAR                                         (* ... die Achsenbeschriftungen *)
     expo,mantisse,zehner : REAL;
     Minus                : BOOLEAN;
  BEGIN
    expo := log(zahl*1.1);
    IF expo < 0.0 THEN expo := expo - 1.0 END;
    expo := real(entier(expo));
    zehner := pwrOfTen(expo);
    mantisse := zahl/zehner;
    IF mantisse <= 1.0 THEN mantisse := 1.0
      ELSIF mantisse <= 2.0 THEN mantisse := 2.0
      ELSIF mantisse <= 4.0 THEN mantisse := 4.0
      ELSIF mantisse <= 5.0 THEN mantisse := 5.0
      ELSIF mantisse <= 6.0 THEN mantisse := 6.0
      ELSIF mantisse <= 8.0 THEN mantisse := 8.0
      ELSE mantisse:=10.0
    END;
    zahl := mantisse*zehner
  END Glaetten;

BEGIN
  anmelden;
  Hintergrund;
  SetzeBereich(9,9, 630,390);
  IF Fehler THEN
    Schreibe(10,210,3,
      "Fehlerhafte Eingabe: nur gleiche, negative oder weniger als 2 Meßwerte");
```

```
  ELSE
    Glaetten(xmax); Glaetten(ymax);              (* Grafikausgabe vorbereiten *)
    SetzeSkalierung(-xmax/10.0, -ymax/10.0, 1.1*xmax, 1.1*ymax);
    AchsenKreuz(xmax/10.0, ymax/10.0, 2, 2, "X", "Y");
    HideCursor(Geraet);
    SetMarkerType(Geraet,starMark);
    FOR i:=0 TO HIGH(Messung) DO PlotW(Messung[i].x, Messung[i].y) END;
    LineW(0.0, Abschnitt, xMax, Anstieg*xmax + Abschnitt);
    Minus := (Anstieg < 0.0);                    (* Textausgabe vorbereiten *)
    Anstieg := ABS(Anstieg);
    s1 := RealToStr(Anstieg,0,4);
    s2 := RealToStr(Abschnitt,0,4);
    s3 := RealToStr(BestimmtMass,0,6);
    s  := "Ausgleichsgerade:  y = ";
    Append(s1,s,ok); Append(" * x ",s,ok);
    IF Minus THEN Append("- ",s,ok) ELSE Append("+ ",s,ok) END;
    Append(s2,s,ok);
    Append("    Bestimmtheitsmaß: ",s,ok); Append(s3,s,ok);
    Schreibe(25,385,2,s);
    ShowCursor(Geraet,FALSE);
  END;
  KeyboardEvent(taste);
  abmelden
END Ausgabe;

BEGIN
  Eingabe;
  Berechnen(Werte,Steigung, yAbschnitt, BMass, xMax, yMax, ok);
  Ausgabe(Werte,Steigung,yAbschnitt,BMass,xMax,yMax,ok);
END MessReihen.
```

Hier nun für Interessierte die mathematische Herleitung für die Steigung m und den Achsenabschnitt b der Ausgleichsgeraden:

Es sei (x_k, y_k) ein Wertepaar, dann beschreibt der Ausdruck $y_k - (m \cdot x_k + b)$ die Abweichung des k-ten Punktes von der Ausgleichsgeraden. Da diese Abweichung sowohl positiv als auch negativ sein kann, betrachtet man als Maß für den Fehler f_k der Einzelmessung das Quadrat dieser Abweichung.
In der Summe

$$F(m,b) = \Sigma f_k = \Sigma (y_k - (m \cdot x_k + b))^2$$

sind also m und b so zu bestimmen, daß die Summe minimal wird. Man sieht, daß F sowohl in m als auch in b eine quadratische Funktion ist.

Wie sich mittels Differentialrechnung oder der Scheitelpunktform der Parabelgleichung ergibt, hat eine quadratische Funktion

$q(z) = a_1 \cdot z^2 + a_2 \cdot z + a_3$ mit $a_1 > 0$

ihr Minimum an der Stelle

$z_{min} = -a_2/(2 \cdot a_1)$.

Nutzt man dies für F(m) und F(b) aus, so erhält man für die Steigung m und den Achsenabschnitt b nach kurzer Rechnung die Beziehungen:

$$m = \frac{\Sigma x_k y_k - \Sigma x_k \cdot \Sigma y_k / n}{\Sigma x_k{}^2 - (\Sigma x_k)^2 / n}$$

$$b = (\Sigma y_k - m \cdot \Sigma x_k) / n$$

wobei jeweils über alle n Wertepaare zu summieren ist. Als Maß dafür, wie gut die Gesamtheit der Punkte die Ausgleichsgerade annähert, benutzt man in der Statistik das sogenannte *Bestimmtheitsmaß* r^2:

$$r^2 = \frac{(\Sigma x_k y_k - \Sigma x_k \cdot \Sigma y_k / n)^2}{(\Sigma x_k{}^2 - (\Sigma x_k)^2 / n)\ (\Sigma y_k{}^2 - (\Sigma y_k)^2 / n)}$$

Die Theorie zeigt, daß für den *Korrelationskoeffizienten* r stets die folgenden Grenzen bestehen: $-1 \leqq r \leqq 1$, also $0 \leqq r^2 \leqq 1$. Liegen alle Punkte exakt auf einer Geraden, so folgt $r^2 = 1$, die Messung ist also um so weniger mit Fehlern behaftet, desto mehr sich r^2 an 1 angleicht. In der Prozedur `Berechnen` werden die obigen Terme für `m`, `b` und `r²` ermittelt. Außerdem wird noch der größte x-Wert und der größte y-Wert festgehalten für die Erstellung der Grafik. Da diese Werte eventuell sehr »krumm« sein können, werden sie in der Prozedur `Glaetten` auf einen »glatten« Wert aufgerundet (für die Einteilungen des Achsenkreuzes). Es wird einschränkend davon ausgegangen, daß alle Meßwerte positiv sind. Ansonsten sind die Prozeduren `Berechnen` und `Ausgabe` allgemein gehalten. Lediglich die Prozedur `Eingabe` ist nur rudimentär als Testprozedur ausgeführt. Es gibt hier je nach Verwendungszweck verschiedene Eingabemöglichkeiten:

- Wenn man nur gelegentlich Messungen kleinerer Versuchsreihen auswerten muß, genügt eine Eingabe der Daten über die Tastatur. Schreiben Sie sich eine redigierbare Eingabeprozedur hierzu (Korrigieren und Hinzufügen von Wertepaaren, nachdem man die Grafik gerechnet hat). Die Techniken dazu wurden in Kapitel 1.7 gezeigt.

- Wenn viele Meßdaten anfallen, so wird man sie direkt von einem externen Gerät einlesen, zum Beispiel von einem Analog-/Digital-Wandler (A/D-Wandler).
- Die Meßwerte liegen als (ASCII-)Text in einem File vor. Das Programm braucht nur das File einzulesen. Dies ist die übliche Methode bei großen Datenmengen! Die Lösung dieser Aufgabe finden Sie auf der Diskette.

Neben einer Verbesserung der Eingabeprozedur sind auch noch folgende Erweiterungen denkbar:

- Man läßt auch negative Werte zu. In diesem Fall muß auch der minimale x-Wert und der minimale y-Wert ermittelt werden.
- Ausgabe eines Meßprotokolls auf den Drucker
- Sortieren der Meßwerte (nach x oder y)

Oft hat man Meßreihen, die keinen linearen Zusammenhang zwischen den Meßgrößen x und y nahelegen, zum Beispiel kann x antiproportional zu y sein (y=m/x+b).

Man kann dann das vorliegende Programm trotzdem nutzen, wenn man statt der Meßwerte x_k einfach $1/x_k$ eingibt. Damit man nicht 1/x zunächst mit einem Taschenrechner ermitteln muß, kann man dies dem Programm überlassen, indem man vor der Berechnung der Ausgleichsgeraden durch ein Untermenü den vermuteten Zusammenhang erfragen läßt.

In die Berechnungen für m, b, und r läßt man dann nicht die Zahlen x_k, sondern die Funktionswerte $g(x_k)$ mit g(x) = 1/x eingehen.

Für andere funktionale Beziehungen kann es auch nötig sein, die y-Werte mit einer Funktion h(y) zu verbiegen. Denkbar sind vor allem folgende Funktionen:

1. Linearer Zusammenhang: $y = m x + b$,
2. Quadratischer Zusammenhang: $y = m x^2 + b$,
3. Antiproportionaler Zusammenhang: $y = m / x + b$,
4. Antiproportional-quadratischer Zus.: $y = m / x^2 + b$,
5. Exponentieller Zusammenhang: $y = k e^{mx}$,
6. Potenzfunktionaler Zusammenhang: $y = a x^b$

Die Ausgabe der Ausgleichsfunktion ist auch entsprechend zu modifizieren. Anleitung: Definieren Sie eine globale Variable art, die im Untermenü mit der Nummer des Zusammenhangs 1 bis 6 belegt wird. Programmieren Sie nun die Funktionen

```
PROCEDURE g(x: REAL): REAL;
PROCEDURE h(x: REAL): REAL;
```

mit einer CASE-Anweisung je nach `art`. In der folgenden Tabelle sind die funktionalen Zusammenhänge wiedergegeben (SQR(x) ist dabei x*x):

1. g(x) = x, h(x) = y
2. g(x) = SQR(x), h(x) = y
3. g(x) = 1/x, h(x) = y
4. g(x) = 1/SQR(x), h(x) = y
5. g(x) = x, h(x) = LN(y)
6. g(x) = LN(x), h(x) = LN(y)

Besonders ergonomisch wäre es, wenn man die verschiedenen Funktionstypen sowie die Punkte Eingabe, Graphik, Druckerausgabe mit Pull-down-Menüs anwählen könnte. Wie man diese programmiert, erfahren Sie im nächsten Abschnitt.

4.7 GEM-Menütechnik und Ereignisbehandlung

Sicherlich ist es Ihnen aufgefallen, daß in professionellen Programmen auf dem Atari Pull-down-Menüs verwendet werden, die eine sehr übersichtliche Benutzerführung gestatten. Pull-down-Menüs werden vom AES automatisch verwaltet, was die Handhabung beim Programmieren deutlich vereinfacht. AES verlangt allerdings, daß das Pull-down-Menü in einer besonderen Datenstruktur – einem »Objektbaum« – vorliegt. In einer solchen Struktur sind auch die Ikonen, die Sie vom Desktop her kennen, abgespeichert; ebenso die Dialogboxen, auf die wir im nächsten Abschnitt zu sprechen kommen. Ein Objektbaum besitzt also eine recht komplexe Struktur, daher ist es aufwendig, ihn selbst zu programmieren. Diese Arbeit wird durch ein Hilfsprogramm, dem sogenannten »Resource-Construction-Set«, einem Editor für Objektbäume, bedeutend erleichert. Ein solches Programm läßt sich ganz preiswert erwerben. Beim Megamax-System wird ein Resource-Construction-Set mitgeliefert. Hiermit kann man bei der Erstellung die Bildschirmoberfläche so vor sich sehen, wie sie später im Programm erscheinen soll. Der Resource-Construction-Set speichert die Objektbäume – also den gesamten Bildschirmzauber aus Menüs, Dialog- und Alarmboxen sowie Ikonen in einer »Resource-Datei« (mit der Endung ».RSC«). Diese Datei kann dann mit der Prozedur `AESResources.LoadResource` in ein Modula-Programm geladen werden. Die Prozedur `AESMenues.MenuBar` bringt das Menü auf den Bildschirm.

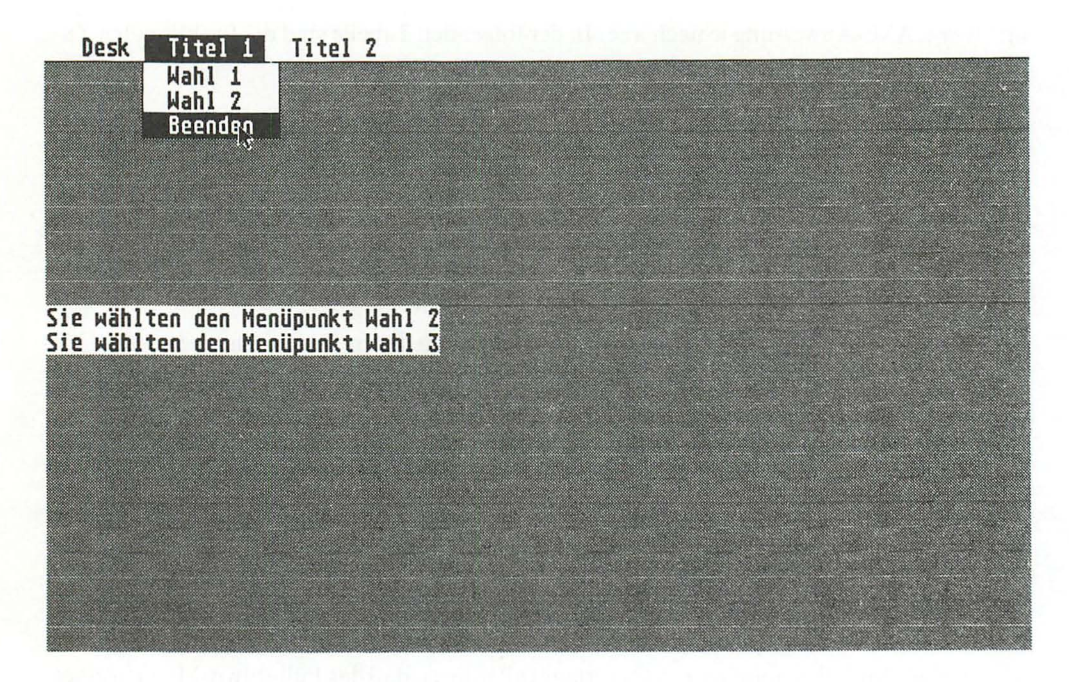


Bild 4.20: Ein Pull-down-Menü

In dem Resource-File sind alle Objekte (Menüpunkte) willkürlich durchnumeriert. Glücklicherweise erzeugt das Resource-Construction-Set einen externen Modul (bestehend aus Implementations- und Definitions-Datei), in dem die Menüpukte als Konstanten definiert sind. Diese Bezeichner wurden zuvor mit dem Resource-Construction-Set deklariert:

```
DEFINITION MODULE Menuel;
CONST
  Menue   =  0;          (* Menuebaum *)
  Titel1  =  4;    (* TITLE in Baum MENUE *)
  Titel2  =  5;    (* TITLE in Baum MENUE *)
  Titel   =  8;   (* STRING in Baum MENUE *)
  Wahl1   = 17;   (* STRING in Baum MENUE *)
  Wahl2   = 18;   (* STRING in Baum MENUE *)
  Ende    = 19;   (* STRING in Baum MENUE *)
  Wahl3   = 21;   (* STRING in Baum MENUE *)
  Wahl4   = 22;   (* STRING in Baum MENUE *)
END Menuel.
```

Der zugehörige Implementationsmodul enthält nur Compileroptionen, ansonsten ist er leer. Es handelt sich also bei `Menuel` um einen reinen Datenmodul (vgl Abschnitt 1.7.5)

```
IMPLEMENTATION MODULE Menue1;
(*$N+*)                  (* Runtime-Modul wird nicht automatisch importiert *)
(*$M-*)                            (* Keine Prozedurnamen im erzeugeten Code *)
END Menue1.
```

Dieser Modul kann ganz normal übersetzt und importiert werden; damit stehen dem Programm dann die Konstanten zur Verfügung.

Um nun das Menü auf den Bildschirm zu bringen, sind folgende Schritte nötig:

1. Anmelden beim GEM wie gewohnt.
2. Das Resource-File, welches (unter anderem) das Menü enthält, von der Diskette in den Speicher laden (mit `LoadResource`)
3. Man besorgt sich einen Pointer auf das Menü. Dies erledigt die Prozedur `ResourceAddr`. Sie benötigt die Nummer, die unser Menü im Resorce-File besitzt: Dafür benötigen wir die Konstante `Menue` aus dem vom RCS erzeugten Modul.
4. Das Menü wird mit `MenuBar` angezeigt.

Die Sequenz im einzelnen:

```
IMPORT Menue1;                              (* Das vom RCS erzeugte Modul *)
<...>
Grafik.anmelden;                                                   (* 1 *)
LoadResource("A:\MENUE1.RSC");                                     (* 2 *)
UnserMenue = ResourceAddr(treeRsrc, Menue1.Menue);                 (* 3 *)
MenuBar(UnserMenue, TRUE);                                         (* 4 *)
```

Das Menü ist nun auf dem Bildschirm. Jetzt muß man nur noch warten, bis der Benutzer einen Menüpunkt anwählt. Dazu ruft man die Prozedur `MessageEvent` (engl. `event`= Ereignis) auf, der man eine Variable vom Typ `MessageBuffer` übergeben muß:

```
VAR Nachricht: AESEvents.MessageBuffer;
<...>
AESEvents.MessageEvent (Nachricht);
```

Enthält nun `Nachricht.msgType` den Wert `menuSelected`, so hat der Benutzer einen Menüpunkt angewählt. Die Nummer des Menüpunktes befindet sich dann in `Nachricht.selItem`.

```
MODULE MenueTest;

FROM Terminal     IMPORT WriteString, WriteLn, KeyPressed, GotoXY;
FROM Grafik       IMPORT anmelden, abmelden;
FROM GEMEnv       IMPORT GemError;
FROM GEMGlobals   IMPORT PtrObjTree;
FROM AESMenus     IMPORT MenuBar, NormalTitle, CheckItem;
FROM AESEvents    IMPORT MessageBuffer, MessageEvent, menuSelected;
FROM AESResources IMPORT LoadResource, FreeResource, ResourceAddr, ResourcePart;
FROM Menuel       IMPORT Wahl1, Wahl2, Wahl3, Ende;
                    (* Menuel enthält die Menüpunktnummern als Konstanten *)

VAR SchalterWahl3 : BOOLEAN;

PROCEDURE p1;
BEGIN
  WriteLn; WriteString("Sie wählten den Menüpunkt Wahl 1")
END p1;

PROCEDURE p2;
BEGIN
  WriteLn; WriteString("Sie wählten den Menüpunkt Wahl 2")
END p2;

PROCEDURE p3(M : PtrObjTree);          (* Menüpunkt mit Häkchen behandeln *)
BEGIN
  SchalterWahl3 := NOT SchalterWahl3;                (* Schalter umschalten *)
  CheckItem(M, Wahl3, SchalterWahl3);             (* Häkchen löschen/setzen *)
  WriteLn;
  IF SchalterWahl3
    THEN WriteString("Sie haben den Menüpunkt Wahl 3 eingeschaltet")
    ELSE WriteString("Sie Haben den Menüpunkt Wahl 3 ausgeschaltet")
END p3;

PROCEDURE ende;
BEGIN
  WriteLn;
  WriteString("Sie wählten den Menüpunkt Ende. Bitte Taste drücken.");
  REPEAT UNTIL KeyPressed()
END ende;

PROCEDURE messageHandler(M : PtrObjTree) : BOOLEAN;
VAR  mb : MessageBuffer;
BEGIN
```

```
  MessageEvent(mb);                            (* Wartet auf ein Ereignis *)
  CASE mb.msgType OF                    (* Welches Ereignis ist eingetreten? *)
    menuSelected :              (* Ereignis: ein Menüpunkt wurde angewählt *)
        CASE mb.selItem OF
          Wahl1 : p1 |
          Wahl2 : p2 |
          Wahl3 : p3 |
          Ende  : ende; RETURN TRUE                               (* fertig *)
        END;
        NormalTitle(M, mb.selTitle, TRUE);            (* Titel wieder normal *)
   ELSE END; (* Die anderen Record-Komponenten sind hier nicht von Belang *)
   RETURN FALSE                                      (* noch nicht fertig *)
END messageHandler;

PROCEDURE arbeiten;
VAR M : PtrObjTree;
BEGIN
  LoadResource("F:\MENUE1.RSC");         (* Resource File ins Programm laden *)
                                    (* Achtung: Pfadnamen geeignet anpassen ! *)
  IF GemError() THEN HALT END;          (* Fehler beim Laden, dann Abbruch *)
  M := ResourceAddr(treeRsrc,Menue);      (* treeRsrc aus Typ ResourcePart *)
  MenuBar(M,TRUE);
  GotoXY(0,10);
  SchalterWahl3 := TRUE;
  REPEAT UNTIL messageHandler(M)
END arbeiten;

BEGIN
  anmelden;
  arbeiten;
  abmelden
END MenueTest.
```

Wie man sieht, ist der Umgang mit Pull-down-Menüs sehr elegant in Modula mit einer CASE-Anweisung zu erledigen. Diese Fallunterscheidung steht in einer Schleife und ruft solange die zugeordneten Prozeduren auf, bis der Benutzer sich für den Menüpunkt »Ende« entschlossen hat.

Wenn Sie im Umgang mit dem Resource-Construction-Set noch unerfahren sind, hier noch ein kleiner Tip, bevor Sie eigene GEM-Bildschirme kreieren: Laden Sie sich zunächst die Resource-Dateien anderer Programme in diesen Editor, und schauen Sie nach, welche Eigenschaften den einzelnen Objekten jeweils mitgegeben wurden, um ihre speziellen Funktionen zu realisieren. Nehmen Sie dazu beispielsweise die Resource-Datei dieses Programms von der

Diskette oder die Ihrer Modula-Shell. Sie sollten aber grundsätzlich mit einer Kopie arbeiten, am besten auf einer RAM-Disk – da schnell im Resource-Construction-Set etwas versehentlich abgeändert ist. Es wäre doch zu schade, wenn ihr Modula-System nicht mehr mit seiner Benutzeroberfläche klar käme!

4.8 Benutzung von Dialogboxen

Zum Abschluß wollen wir unseren Programmen noch den richtigen »Atari-Pep« geben: Bei unseren vorherigen GEM-Beispielen ging es in erster Linie um Ausgaben. Eingaben erfolgten bisher nur mit Mausklick (Eingabe von Punkten/Koordinaten) oder über die Knöpfe einer Alertbox. Ab und zu kommt man aber nicht umhin, auch in einem GEM-Programm zur Tastatur zu greifen, um Strings oder Zahlen einzugeben. Hierzu verwendet man »Dialogboxen«. Das Aussehen (Größe, Lage auf dem Bildschirm, Masken (der feste Text), Art der Eingaben (Buchstaben oder Ziffern) legt man am einfachsten mit dem Resource-Construction-Set fest. Wie bei den Menüs liest man dann im Modula-Programm die Resource-Datei mit der fertigen Dialogbox ein. Mit `DrawObject` wird die Box auf den Bildschirm gebracht. Der Benutzer kann nun in Ruhe seine Eingaben machen und beliebig verbessern und edieren. Erst wenn er mit dem Tippen fertig ist (was er im allgemeinen durch Anklicken eines mit »OK« beschrifteten Knopfes meldet), erhält das Programm die Kontrolle (und die ausgefüllte Dialogbox) zurück.

Bild 4.21: Eine Dialogbox

Wie bringt man nun eine Dialogbox, die man mit dem RCS erzeugt hat, von seinem Programm aus auf den Bildschirm? Kein Problem, wenn man sich genau an das folgende Rezept hält:

1. Wie immer: Beim GEM anmelden. Am einfachsten natürlich mit unserer Prozedur `Grafik.anmelden`.
2. Das Resource-File, das die Dialogbox enthält, mit `LoadResource` von der Diskette in den Speicher laden. Das Resource-File kann noch weitere Dialogboxen oder Menüs enthalten. **Wichtig:** Es darf nur EIN Resource-File geladen werden.
3. Man besorgt sich einen Pointer auf die Dialogbox.
4. Den Bereich, den die Box auf dem Bildschirm belegt, (mit `FormDial(reserveForm,...)` reservieren. Dazu benötigen wir die Größe der Dialogbox (als Rechteck), welche die Funktion `FormCenter` liefert.
5. Man kann mit `FormDial(growForm,...)` den bekannten »Zoom«-Effekt beim Aufgehen von Fenstern erzeugen. Dazu wird ein kleineres Rechteck (in unserem Beispiel `boxklein`) benötigt, das dann auf die Größe von `box` »gezoomt« wird. Dieser Schritt ist mehr ein Gag und kann entfallen.
6. Die Box mit `DrawObjekt` auf den Bildschirm bringen.
7. Den Dialog einleiten mit `FormDo`. Die Box steht nun dem Anwender zur Verfügung, bis sie durch Anklicken des OK- oder Abbruch-Knopfes verlassen wird. Die Nummer des Knopfes, mit dem der Benutzer den Dialog beendet hat, steht im 3. Parameter von `FormDo`.
8. Für den Abgang der Box kann wieder ein »Zoom-Effekt« erzeugt werden, diesmal in umgekehrter Richtung: `FormDial(shrinkForm,boxklein,box)`. Anmerkung: auch hier steht das kleinere Rechteck zuerst in der Parameterliste.
9. Die Box wieder abmelden: `FormDial(freeForm,...)`

Anmerkung: die Schritte 1 bis 3 sind nur einmal im gesamten Programm erforderlich.

Für einen erneuten Dialog mit der Box sind die Schritte 4 bis 9 zu wiederholen.

```
IMPORT Dialog1;                        (* das vom RCS erzeugte Datenmodul *)
<...>
VAR
    UnserDialog: GEMGlobals.PtrObjTree;
    box,                                     (* für die Größe der Box *)
    boxklein: Rectangle;                      (* für den ZOOM-Effekt *)
<...>
Grafik.anmelden;
AESResources.LoadResource("DIALOG1.RSC");                       (* 2 *)
```

```
UnserDialog := ResourceAddr(treeRsrc, Dialog);                      (* 3 *)
<...>
FormDial(reserveForm, boxklein, box);                                (* 4 *)
FormDial(growForm,    boxklein, box);                                (* 5 *)
DrawObject(UnserDialog, Root, MaxDepth, box);                        (* 6 *)
FormDo    (UnserDialog,Root,   EndeKnopf);                           (* 7 *)
FormDial(shrinkForm,  boxklein, box);                                (* 8 *)
FormDial(freeForm,    boxklein, box)                                 (* 9 *)
```

Nun tritt die Schwierigkeit auf, an die eingegebenen Daten heranzukommen. Hierzu gibt es zwei Möglichkeiten:

1. Wir sagen der Dialogbox, wohin sie die Eingaben schreiben soll. Das geschieht mit der Prozedur `ObjHandler.LinkTextString`, deren erster Parameter die Nummer des Feldes in der Dialogbox ist. Diese erhält man aus dem Resource-Construction-Set erzeugten Definitionsmodul. Der zweite Parameter ist die Adresse einer String-Variablen, die die Daten aufnehmen soll. Alle Anwendereingaben in der Dialogbox gelangen also direkt ans richtige Ziel. Zu beachten ist noch, daß sämtliche Eingaben von Zeichenketten aufgenommen werden, Zahlen müssen nachträglich umgewandelt werden. Wichtig ist hierbei, daß zuvor bei der Arbeit mit dem Resource-Construction-Set die maximal benötigte Stringlänge korrekt vordefiniert worden ist. Sie muß auch unbedingt mit der Länge des aufnehmendenden Strings übereinstimmen. Bei einem kürzeren Zielstring könnte es andernfalls passieren, daß ungewollt Speicherbereich überschrieben wird, denn die Prozedur `LinkTextString` kennt ja nur dessen Adresse. Es findet also keine Bereichskontrolle statt, die Programmiersprache »C« läßt grüßen!

2. Ohne besondere Vorkehrungen werden die Eingaben in die Dialogbox automatisch im Objektbaum selbst gespeichert. Er hält den nötigen Speicherplatz hierfür bereit. Man braucht nur noch den String in eine Variable des Klientenprogramms zu kopieren. Dies leistet die Prozedur `ObjHandler.GetTextStrings`. Ihre Parameter sind Nummer des betreffenden Feldes in der Dialogbox und der Zielstring. Die beiden weiteren Parameter enthalten die zuvor im Resource-Construktion-Set festgelegten Stringmasken und sind hier nicht von Belang.

Zunächst folgt wieder zum besseren Verständnis der vom Resouce-Construktion-Set erzeugte Datenmodul:

```
DEFINITION MODULE Dialog1;

CONST
  Dialog   =   0;          (* Formular/Dialog *)
```

```
  Titel    =  1;  (* BOXTEXT in Baum DIALOG *)
  Nachname =  3;    (* FTEXT in Baum DIALOG *)
  Vorname  =  4;    (* FTEXT in Baum DIALOG *)
  Alter    =  5;    (* FTEXT in Baum DIALOG *)
  Quit     =  6;   (* BUTTON in Baum DIALOG *)
  Ok       =  7;   (* BUTTON in Baum DIALOG *)
END Dialogl.
```

```
IMPLEMENTATION MODULE Dialogl;
(*$N+,M-*)
END Dialogl.
```

Das Demonstrationsprogramm zeigt beide besprochenen Möglichkeiten, die Eingaben »abzuholen«. Damit Sie sehen, daß alles korrekt funktioniert, werden die geänderten Strings anschließend einfach auf den Bildschirm geschrieben.

```
MODULE DialogTest;

FROM SYSTEM       IMPORT ADDRESS, ADR;
FROM InOut        IMPORT WriteString, WriteLn, Read;
FROM Grafik       IMPORT anmelden, abmelden;
FROM GEMGlobals   IMPORT PtrObjTree, Root, MaxDepth, OStateSet;
FROM GEMEnv       IMPORT GemError;
FROM GrafBase     IMPORT Rectangle, Rect;
FROM ObjHandler   IMPORT LinkTextString, GetTextStrings,
                         SetCurrObjTree, SetObjState;
FROM AESObjects   IMPORT DrawObject;
FROM AESForms     IMPORT FormCenter, FormDial, FormDo, FormDialMode;
FROM AESResources IMPORT LoadResource, FreeResource, ResourceAddr,
                         ResourcePart;
FROM Dialogl      IMPORT Dialog,Nachname,Vorname,Alter,Ok,Quit;

TYPE
    str80 = ARRAY[0..79] OF CHAR;

VAR
    dlogstr  : RECORD
                 Name, Vorname : str80;
                 Alter         : str80
               END;
     dlogBaum : PtrObjTree;
```

```
PROCEDURE ResourceBauen;
BEGIN
  LoadResource("DIALOG1.RSC");          (* Resource File ins Programm laden *)
  IF GemError() THEN HALT END;          (* Fehler beim Laden,  dann Abbruch *)
  dlogBaum := ResourceAddr(treeRsrc, Dialog);
  SetCurrObjTree(dlogBaum,FALSE);
  LinkTextString(Nachname, ADR(dlogstr.Name));              (* 1. Möglichkeit *)
  LinkTextString(Vorname, ADR(dlogstr.Vorname));
  dlogstr.Name := "B.L. Ödmann";                       (* Strings vorbesetzen *)
  dlogstr.Vorname := ""
END ResourceBauen;

PROCEDURE BoxZeigen(baum: PtrObjTree): CARDINAL;
VAR
    boxklein, box : Rectangle;
    EndeKnopf     : CARDINAL;
BEGIN
    boxklein := Rect(200,200, 50,30);
    box := FormCenter(baum);
    FormDial(reserveForm, boxklein, box); (* Bildschirmbereich reservieren *)
    FormDial(growForm, boxklein, box);       (* Effekt: Zoom klein --> groß *)
    DrawObject(baum, Root, MaxDepth, box);         (* Dialogbox zeichnen *)
    FormDo(baum, Root, EndeKnopf);                   (* Benutzereingaben *)
    FormDial(shrinkForm, boxklein, box);     (* Effekt: Zoom groß --> klein *)
    FormDial(freeForm, boxklein, box);       (* Bildschirmbereich freigeben *)
    RETURN EndeKnopf
END BoxZeigen;

PROCEDURE arbeiten;
VAR
    c     : CHAR;
    dummy : str80;
BEGIN
  ResourceBauen;                        (* Resourcen laden und vorbelegen *)
  WHILE BoxZeigen(dlogBaum) = Ok DO
       SetCurrObjTree(dlogBaum, FALSE);
       GetTextStrings(Alter, dlogstr.Alter, dummy,dummy);       (* 2. Mögl. *)
       WITH dlogstr DO
             WriteLn; WriteString("Sie heißen: ");
             WriteString(Vorname); WriteString(" "); WriteString(Name);
             WriteLn; WriteString("Ihr Alter: "); WriteString(Alter)
       END;
```

```
        SetObjState(Ok, OStateSet{});  (* OK-Taste wieder abschalten *)
    END;
    WriteLn; WriteString("Das wars dann wohl.... <Taste>");
    Read(c);
END arbeiten;

BEGIN
    anmelden;
    arbeiten;
    abmelden
END DialogTest.
```

Das Programm soll nur die Programmiertechnik von Dialogboxen verdeutlichen; ansonsten hat es keinen eigenständigen Wert. Im Kapitel 5 sehen Sie aber das Zusammenspiel von Dialogboxen, Menüleisten, Alertboxen und Grafik in einer komplexen Anwendung.

4.9 Benutzung des SSWiS-Moduls bei SPC-Modula

Wie eingangs erwähnt, wurden die GEM-Beispiele dieses Buchs mit dem Megamax-Modula-System entwickelt. Sie lassen sich aber auf jedes andere Modula-System für den Atari ST übertragen, da diese sämtlich über Module mit AES- und VDI-Routinen verfügen. Nun wollen wir hier noch auf eine Eigenheit, besser gesagt Spezialität, von SPC- Modula eingehen.

Ein Teil unserer Beispiele zeigte, daß GEM-Programme auf Vorarbeiten mit einem Resource-Construction-Set basieren. Dies wird nicht als Nachteil erscheinen, wenn man eine komplexe GEM-Umgebung in einem Programm wünscht, da sich alle Elemente (Menüzeilen, Dialog- und Alertboxen, Programmlogo) in einem Arbeitsgang erzeugen lassen. Für eine kleinere Anwendung, in der man beispielsweise nur eine Dialogbox und ein Fenster benötigt, ist es jedoch etwas umständlich.

Bei SPC-Modula gibt es einen Modul »SSWiS« (= **S**mall **S**ystems **Wi**ndowing **S**tandard) mit dem sich einfach fensterorientierte Programme schreiben lassen. Die Verfasser erhoffen sich von SSWiS zudem eine Verbreitung auf andere Rechner, so daß eine systemunabhängige einheitliche Fensterschnittstelle bereit stünde, die die Portierung von Modula-Programmen auf andere Rechner weiter vereinfachen würde.

Neben einer einfach zu bedienenden Fensterverwaltung bietet SSWiS eine handliche Möglichkeit zur Erzeugung von Alert- und Dialogboxen. Die SSWiS-Alertboxen – sie heißen hier

Notizboxen – sind denen des GEM recht ähnlich. Sie besitzen zwar kein Icon, gestatten dafür aber bis zu vier Knöpfe.

Die SSWiS-Dialogboxen haben ebenfalls eine standardisierte Form und lassen sich praktisch mit einem einzigem Prozeduraufruf bewerkstelligen. Sie bestehen aus einer Meldungstextzeile und einer edierbaren Zeile. Neben den gewohnten Quittungtasten zur Dialogbeendigung kann man noch »Optionstasten« für Voreinstellungen programmieren, die durch Anklicken mit der Maus an- bzw. abgeschaltet werden können.

Eine besondere Form des Dialogs stellt das »Identifikationsformular« dar, das bei SSWiS anstelle des Programmlogos tritt. Die Identifakationsbox erscheint nach Anwahl des ersten Menüeintrags der ersten Menüspalte. Das Pulldown-Menü hat unter SSWiS das übliche Aussehen und Bedienungsart. Es kann vom Programm aus mit wenigen Prozeduraufrufen erzeugt werden.

Besonders zu erwähnen ist noch, daß unter SSWiS mehrere Anwendungen quasiparallel ablaufen können. Ein gutes Beispiel hierfür ist die Shell des SPC-Systems, in der man die Ausgabe eines Programms in einem Fenster gleichzetig mit dem Quelltext in einem anderen Fenster sehen kann. Jede SSWiS-Anwendung erhält ihren eigenen Menübalken. Es können also mehrere Menüs verwaltet werden. Sichtbar ist immer das Menü, das zu der Anwendung gehört, deren Fenster momentan »aktuell« ist. Das aktuelle Fenster ist dabei das oberste Fenster.

Alles bisher Gesagte soll nun in einem kleinen Programm »SSWiSDemo« gezeigt werden, also Menüleiste, Dialogbox, Notizboxen, Identifikationsbox und ein Fenster.

Das Menü hat den Titel »Kommando« mit den Einträgen »Satz eintragen«, »Alertbox«, »Ende«. Beim Anklicken von »Satz eintragen« erscheint eine Dialogbox zur Eingabe eines Strings. Dieser String wird dann in eine zufällige Zeile eines Textfensters geschrieben. SSWiS selbst stellt nur die reine Fensterverwaltung dar, zur Ausgabe von Text oder Grafik gibt es hier keine Prozeduren. Zur Textausgabe benötigt man Prozeduren aus dem Modul »TextWindows«. TextWindows organisiert ein Fenster statt in »Weltkoordinaten« in Textzeilen und -spalten.

Klickt man in der äußersten linken Menüleiste »SSWiS Demo« an, so wird die Identifikationsbox geöffnet. Das Programm terminiert durch die <Esc>-Taste oder Anwahl des Menüpunktes »Ende«.

Während des Programmablaufs kann man die Größe und Lage des Textfensters mit der Maus manipulieren. Das Textfenster läßt sich auch schließen, wodurch automatisch unsere Menüzeile verschwindet. Da unser Fenster jetzt nicht mehr aktuell ist, erscheint das Menü von SSWiS, das die Anwahl des SPC-Programmlogos gestattet. Unser Fenster läßt sich aber jederzeit wieder öffnen. Hierzu befindet sich am unteren linken Bildschirmrand ein Rechteck mit dem Namen der Anwendung. Durch Anklicken dieses Rechteckbereichs lebt sie wieder auf.

Diesen ganzen Bildschirmzauber verwaltet SSWiS selbst, er braucht also nicht explizit programmiert zu werden.

Was bleibt nun noch für den Programmierer einer SSWiS-Anwendung zu tun? Grob gesagt, geht es um das Anmelden der Anwendung beim SSWiS, wobei einige Arbeitsprozeduren übergeben werden müssen, gefolgt vom Aufruf einer Arbeitsschleife und schließlich vom Abmelden. Die Arbeitsprozeduren hat der Programmierer bereitzustellen.

Im einzelnen zeigt das Hauptprgramm von SSWISDemo die typische Vorgehensweise:

1. Die Anwendung meldet sich mit `SSWiS.Register(Client,"SSWiS Demo",Accept)` als »Kunde« beim SSWiS an (Prozedur Init), indem dort eine Prozedur `Accept` übergeben wird, die auf alle Benutzertätigkeiten reagieren soll. Sie entspricht der Prozedur `messageHandler` aus dem Abschnitt 4.7. Sie reagiert also auf Ereignisse wie Tastendruck, Mausclick, Menüanwahl, Timerevents usw. in einer `CASE`-Anweisung. In unserem Fall wird von hier aus der Prozeduraufruf `DialogDemo` oder `AlarmDemo`, das Erscheinen der Identifikationsbox und das Setzen des Exitflags zur Programmbeendigung bewirkt. Verallgemeinert ist die Accept-Prozedur also diejenige Routine, mit der SSWiS später arbeiten soll.

2. In der Prozedur `Open` erfolgt die Initialisierung des benötigten Textfensters durch `TextWindows.Create(Client,Fenster,Restore)`. Bei der Ereignisbehandlung kann es vorkommen, das Bereiche des Fensters restauriert (neu geschrieben) werden müssen. Dazu benötigt SSWiS eine Prozedur `Restore`. Die Restore-Prozedur wird von SSWiS automatisch aufgerufen, wenn ein Neuzeichnen erforderlich ist, beispielsweise nach dem Schließen einer Dialogbox, die das Fenster teilweise verdeckte. Sie muß dann in der Lage sein, das Fenster (oder Teile davon) zu rekonstruieren. Dazu muß sie den gesamten Inhalt des Fensters kennen. In unserem Programm erreichen wir das durch ein globales Feld `Zeilen`, in dem alle Zeilen des Fensters als String gespeichert sind.
   Weiterhin werden in `Open` die Mindest-, Ideal- und Maximalgröße des Fensters, seine Bedienungselemente und die Menüeinträge definiert. Anschließend öffnet man das Fenster in Idealgröße und das Menü erscheint automatisch.

3. Sobald diese Initialisierungen durchgeführt sind, ruft man `SSWiS.PollEvents` solange auf, bis der Benutzer den Programmabbruch auslöst (Prozedur `Arbeiten`). Hierdurch wird die Ereignisbehandlungsschleife ausgeführt.

4. Mit der Prozedur `Close` wird das Fenster wieder geschlossen.

5. Die Prozedur `Term` meldet dann die Anwendung beim SSWiS wieder ab.

Das Programm SSWISDEM.PRG findet sich auf der Diskette 2 im Ordner SPC. Hier gibt es auch das Quellfile (mit der Endung .MOD, im Gegensatz zu Megamax-Modula, wo Quellfiles auf .M enden!) und das übersetzte, aber ungelinkte Objektfile. Zusätzlich ist noch die Datei

SSWISS.RSC nötig, die von SSWiS automatisch geladen wird. Sie wird vom SPC-Hersteller mitgeliefert. Der Hersteller beabsichtigt, in einer neuen Version diese Datei gänzlich verschwinden zu lassen und die Daten in den SSWiS-Modul zu integrieren.

Am besten, Sie lassen das Programm gleich laufen und testen alles am Rechner aus, während Sie das Quellfile studieren. Die restlichen Fragen dürften sich dann klären, zumal der Text besonders ausführlich kommentiert wurde. Bleibt noch zu erwähnen, daß die Vorlage für dieses Programm von Andreas Gauger (von advanced applications Viczena) stammt, wofür wir ihm hiermit danken.

```
MODULE SSWiSDemo;

IMPORT SSWiS,TextWindows,Strings,Clock;

CONST
        MaxZeilen  = 50;              ( * Maximale Höhe des Textfensters * )
        MaxSpalten = 79;            ( * Maximale Breite des Textfensters * )

TYPE
        INDEX      = INTEGER;
         (* Bei SPC sind Feldindices vom Typ INTEGER; ebenso HIGH(feld) * )
        Zeile      = ARRAY [0..MaxSpalten] OF CHAR;
                                 (* Typdeklaration für eine Bildschirmzeile * )

VAR
        Zeilen    : ARRAY [0..MaxSpalten] OF Zeile;    (* Feld von Zeilen * )
        Client    : SSWiS.ModuleHandles;          (* SSWiS-Client-Kennung * )
        Fenster   : SSWiS.WindowHandles;               (* Fenster-Kennung * )
        ExitFlag  : BOOLEAN;                      (* Flag für Programmende * )

(* Die Restore-Prozedur wird vom SSWiS vollautomatisch aufgerufen, wenn
 *  ein Teil des Fensters neu aufgebaut werden soll, SSWiS übergibt den
 *  Bereich, der regeneriert werden muß, in Clipxy und Clipwh.
 * )
PROCEDURE Restore(
             Owner   : SSWiS.ModuleHandles;       (* SSWiS-Client-Kennung * )
             Fenster : SSWiS.WindowHandles;            (* Fenster-Kennung * )
             Clipxy,
             Clipwh  : TextWindows.Points);  (* neuzuzeichnedes Rechteck * )

VAR
    y          : INDEX;                                  (* Zeilenzähler * )
    Ausschnitt : Zeile;          (* Zwischenspeicher für die neue Zeile * )
```

```
    Position   : TextWindows.Points;          (* Positions-Zwischenspeicher * )

BEGIN
  Position.X:=Clipxy.X;              (*  X-Position des zu zeichnenden Rechtecks * )
  FOR y := Clipxy.Y TO Clipxy.Y+Clipwh.Y DO (*  alle Zeilen des  Rechtecks * )
    Position.Y := y;                         (*  Y-Position der momentanen Zeile * )
    TextWindows.Position(Position);
    (*  Positioniere TextWindow's interne Schreibmarke
    ...Fehler im Handbuch: nicht der Caret wird positioniert * )
    Strings.Copy(Zeilen[y],Clipxy.X,Clipwh.X,Ausschnitt);
                  (* Benötigten Ausschnitt aus dem Zeilenfeld ausschneiden * )
    TextWindows.WriteString(Ausschnitt);          (*  ... und dann ausgeben * )
  END;
  (* Man könnte auch die ganze Zeile ausgeben, SSWiS fängt alle
   * Zeichen außerhalb des Rechtecks ab, aber es dauert länger.
   * )
END Restore;

(*  liefert eine Zufallszahl zwischen 0 und bereich-1: * )
PROCEDURE Zufall(bereich: LONGINT) : INDEX;
VAR
    SystemZeit : Clock.Time;
BEGIN
    Clock.Get(SystemZeit);                    (*  Holt die aktuelle Uhrzeit * )
    RETURN (SystemZeit.Millisec DIV 5D) MOD bereich
END Zufall;

(*  Die Prozedur DialogDemo erfragt einen Satz vom Benutzer, der
 *  dann in einer zufällig gewählten Zeile des Fensters dargestellt wird.
 *  Diese Routine demonstriert den wichtigen 'ExplicitRestore'-Aufruf, der
 *  die einzige erlaubte Möglichkeit darstellt, Bereiche im Fenster vom
 *  Programm aus neuzuzeichnen
 * )
PROCEDURE DialogDemo;
VAR
        Eingabe : Zeile;                          (* Eingegebener Satz * )
        Options : BITSET;                                     (* Dummy * )
        Knopf   : INTEGER;                       (* Gedrückter Exit-Knopf * )
        y       : INDEX;
        Restxy,                (* Übergabevariablen für ExplicitRestore * )
        Restwh  : TextWindows.Points;
BEGIN
  Eingabe := "";                               (* String 'leer machen' * )
  Knopf := 0;                                  (*  Default-Knopf setzen * )
```

```
  SSWiS.AskForm (* SSWiS-Frage-Formular mit Exit-Knöpfen Fertig & Abbruch * )
     ("Dialogbox-Demo: Geben Sie einen Satz ein!",
      "Fertig|Abbruch", "", Eingabe, Options, Knopf);
  IF Knopf=0 THEN                   (* Wenn der Fertig-Knopf gedrückt wurde * )
      y := Zufall(MaxZeilen);            (* Zufallszeile für Ausgabe wählen * )
      Strings.Copy(Eingabe,0,Strings.Length(Eingabe),Zeilen[y]);
                                (* Den eingegebenen Satz ins Zeilen-Feld kopieren * )
      Zeilen[y][Strings.Length(Eingabe)]:=' ';  (*  String-Ende-Marke weg * )
(*  Die y-te Zeile des Textes muß neu geschrieben werden. Dafür werden die
 *  Koordinaten des neuzuzeichnenden Rechtecks ("Clipping-Rechteck") gesetzt
 *  und anschließend 'TextWindows.ExpilicitRestore' aufgerufen: * )
    Restxy.X:=0;                        (* 'Clipping'-Rechteck definieren... * )
    Restxy.Y:=y;
    Restwh.X:= MaxSpalten+1;
    Restwh.Y:=1;
    TextWindows.ExplicitRestore(Client,Fenster,Restxy,Restwh);
  END;
END DialogDemo;

(*  Diese Prozedur demonstriert die Alert-Boxen.
 * )
PROCEDURE AlarmDemo;
VAR
    Knopf : INTEGER;                               (* gedrückter Exit-Knopf * )
BEGIN
  Knopf := 0;                       (* Default-Knopf ist der erste von links * )
  SSWiS.NotifyForm                                (* Alert-Box darstellen * )
   ("Dies ist eine 'Notiz-Box'","Warum?|Notiz?|Autor?",Knopf);
               CASE Knopf OF
    0:                                       (* 1. Knopf von links gedrückt * )
      Knopf:=-1;                             (* diesmal kein Default-Knopf * )
      SSWiS.NotifyForm                          (* und noch eine Alertbox * )
          ("Nur zur Demonstration","Genial!",Knopf) |
    1:                                       (* 2. Knopf von links gedrückt * )
      Knopf:=-1;
      SSWiS.NotifyForm
        ("SPC Entwickler machen aus Alert->Notiz", "Soso", Knopf) |
    2:                                      (*  3. Knopf von links gedrückt * )
      Knopf:=-1;
      SSWiS.NotifyForm
       ("Der Autor heißt: siehe Menü 'Demo'", "Danke", Knopf)
  ELSE
  END;
END AlarmDemo;
```

```
(* Die Accept-Prozedur wird vom SSWiS aufgerufen, wenn ein Ereignis aufge-
 * treten ist, z.B. eine Maustaste oder eine normale Taste wurde gedrückt,
 * oder ein Menüpunkt wurde angewählt usw.
 * )
PROCEDURE Accept(Owner   : SSWiS.ModuleHandles;   (* SSWiS-Klient-Kennung * )
                 Fenster : SSWiS.WindowHandles;        (* Fenster-Kennung * )
             VAR SReport : SSWiS.EventReports); (* aufgetretenes Ereignis * )
BEGIN
  WITH SReport DO
    CASE Type OF
      SSWiS.Keyboard:                      (* Wenn eine Taste gedrückt wurde * )
        CASE Strokes.Keys[0] OF
          27: ExitFlag:=TRUE                    (*  Esc-Taste => Programmende * )
          ELSE
        END |
      SSWiS.Mouse:                     (* Wenn eine Maustaste gedrückt wurde * )
          (* Dann passiert in diesem Programm gar nix * ) |
      SSWiS.Menu:                      (* Wenn ein Menüpunkt angewählt wurde * )
        CASE Selection.Title OF
          0:      (* Wenn es ein Menüpunkt im zweiten Menü von links war * )
             CASE Selection.Item OF
               0: DialogDemo |         (* 1. MenüPunkt => DialogDemo    * )
               2: AlarmDemo  |         (* 3. MenüPunkt => AlarmDemo     * )
               4: ExitFlag:=TRUE       (* 5. MenüPunkt => Programmende * )
               ELSE
             END
          ELSE
        END |
      SSWiS.Identification:          (* Programm-Identifikation angefordert * )
        SSWiS.Identify(
                   "SSWiS-Demonstrationsprogramm",             (* Prog-Name * )
                   "Aus 'Modula-2 für den Atari ST'",            (* Version * )
                   "Dürholt / Schnur",                            (*  Autor * )
                   "Markt&Technik Verlag");                    (* Copyright * )
      ELSE
    END
  END
END Accept;

(* Diese oder ähnliche Prozeduren werden benutzt, wenn man die Kontrolle
 * an SSWiS übergeben will.
 * )
PROCEDURE Arbeiten;
BEGIN
```

```
  ExitFlag := FALSE;                            (* Programmende-Flag löschen * )
  WHILE NOT ExitFlag DO           (* Solange nicht 'ExitFlag' gesetzt ist ... * )
    SSWiS.PollEvents                 (* ... lassen wir SSWiS für uns arbeiten * )
  END
END Arbeiten;

    (* Initialisiert das Textfenster und setzt die Menüleiste * )
PROCEDURE Open;
VAR
    ScrPos, ScrMinWh, ScrNorWh, ScrMaxWh  : SSWiS.ScreenPoints;
    Worldxy, Worldwh                      : TextWindows.Points;

BEGIN
  TextWindows.Create(Client, Fenster, Restore);
      (* Fenster initialisieren, die Restore-Prozedur wird mit übergeben * )
  SSWiS.SetWindowElements(
              Client, Fenster,
              SSWiS.SetOfWindowElements{  (* alle Fensterelement setzen * )
                  SSWiS.Iconiser, SSWiS.MessageLine,
                  SSWiS.XScroller, SSWiS.YScroller,
                  SSWiS.Sizer, SSWiS.Fuller} );
  SSWiS.SetWindowTitle(Client, Fenster, "SSWiS-Fenster"); (* Titel setzen * )
  SSWiS.SetWindowMessage(Client, Fenster, "Dies ist ein SSWiS-Fenster");
                                                    (* Infozeile setzen * )
  ScrPos.X    := 10;        (* Ausmaße des dargestellten Fensters in Punkten * )
  ScrPos.Y    := 20;
  ScrMinWh.X := 80;
  ScrMinWh.Y := 32;
  ScrNorWh.X := 320;
  ScrNorWh.Y := 160;
  ScrMaxWh.X := 640;
  ScrMaxWh.Y := 400;
  SSWiS.PositionWindow(Client,Fenster,ScrPos);    (* Fenster positionieren * )
  SSWiS.SizeWindowContent(Client,Fenster,ScrMinWh,ScrNorWh,ScrMaxWh);

             (* minimale, normale und maximale Fensterausdehnung setzen * )
  Worldxy.X := 0;    (* Echte Ausmaße des Fensters in Zeichen (TextWindow) * )
  Worldxy.Y := 0;
  Worldwh.X := MaxSpalten;
  Worldwh.Y := MaxZeilen;
  TextWindows.PositionWorld(Client,Fenster,Worldxy);
                              (* Fenster innerhalb der 'Welt' positionieren * )
  TextWindows.SizeWorld(Client,Fenster,Worldwh);   (* 'Welt'-Größe setzen * )
  SSWiS.PlaceWindowOnTop(Client,Fenster);             (* Fenster darstellen * )
```

```
  SSWiS.SetMenuTitle(Client,0," |Kommando");                    (* Menütitel setzen * )
      (* Menüpunkte setzen. M=Maske: nicht anwählbar |(z.B. für Striche): * )
  SSWiS.SetMenuItem(Client, 0,0, " |Satz eingeben");
  SSWiS.SetMenuItem(Client, 0,1, "M|------------");
  SSWiS.SetMenuItem(Client, 0,2, " |Alertbox");
  SSWiS.SetMenuItem(Client, 0,3, "M|------------");
  SSWiS.SetMenuItem(Client, 0,4, " |Ende")
END Open;

 (* Diese Prozedur schließt das Fenster bei Programmende * )
 PROCEDURE Close;
 BEGIN
   TextWindows.Delete(Client, Fenster)
 END Close;

 (* SSWiS-Applikation registriert und Accept-Prozedur übergeben * )
 PROCEDURE Init;
 BEGIN
   SSWiS.Register(Client,"SSWiS-Demo",Accept)
 END Init;

 (*  Hier wird die SSWiS-Applikation abgemeldet * )
 PROCEDURE Term;
 BEGIN
   SSWiS.Deregister(Client)
 END Term;

 (*  Diese Prozedur füllt das Zeilen-Feld 'Zeilen' mit Leerzeichen * )
 PROCEDURE AlleZeilenLoeschen;
 VAR
     i : INDEX;                                              (* Zeilenzähler * )
 BEGIN
   FOR i := 0 TO MaxZeilen DO                                 (* Für alle Zeilen * )
     Zeilen[i] := "";                                    (* Erst mal 'leer machen' * )
    Strings.Pad(Zeilen[i],MaxSpalten," ")      (* und mit Leerzeichen füllen * )
   END
 END AlleZeilenLoeschen;

 BEGIN
   AlleZeilenLoeschen;                          (* Zeilenfeld mit Leerzeichen füllen * )
   Client:=1;                  (* vorsichtshalber mal setzen wegen Compiler-Fehler * )
   Fenster:=1;
   Init;                                                  (* SSWiS Initialisieren * )
   Open;                                            (* Fenster öffnen, Menü setzen * )
```

```
    Arbeiten;                                    (* SSWiS starten * )
    Close;                                    (* Fenster schließen * )
    Term;                                   (* Beim SSWiS abmelden * )
END SSWiSDemo.
```

Wenn Sie mit SPC-Modula arbeiten und weiter in die SSWiS-Progammierung einsteigen wollen, sei noch das Studium der Beispielprogramme »Terminal« (SSWiS-Version) und »Watch« aus dem SPC-Handbuch empfohlen. »Watch« zeigt die Programmierung der Analoguhr, die in der SPC-Shell erscheint. Hier handelt es sich um ein SSWiS-Fenster mit Grafik. Wie oben erwähnt, baut SSWiS beim Atari auf GEM auf, daher ist der Aufruf von AES- und VDI-Routinen in einer SSWiS-Anwendung durchaus angezeigt.

KAPITEL 5

Demonstration der Entwicklung eines komplexen Programmpaketes unter Modula-2

Zu dieser Thematik haben wir uns für ein Programm zum Zeichnen von Funktionsgraphen entschieden. »Schon wieder ein Plotprogramm. Das habe ich doch schon vor Jahren in Basic programmiert« wird da mancher Leser denken. Doch wir meinen, daß wir mit unserem Paket auch dem versierten Programmierer Einiges bieten können:

- Eingabe des Funktionsterms als Zeichenkette mit anschließendem »Scannen« und »Parsen«. Hier wird aber nicht zum hundertsten Mal der Standard-Wirth-Parser portraitiert, sondern es handelt sich um eine Eigenentwicklung, die aus der Zeichenkette einen »Funktionsbaum« erzeugt. Alle Funktionen der `MathLib` werden unterstützt.
- Bestimmung der Ableitung der eingegebenen Funktion, aber nicht einfach numerisch durch Approximation der Tangentensteigung, sondern als Funktionsterm, also algebraisch. Dabei wird gezeigt, wie man dem Computer Ableitungsregeln beibringen kann! Der erhaltene Term wird dabei nach gewissen Regeln optimiert. Das Regelwerk ist beliebig erweiterbar, wir benutzen hierbei Methoden der KI-Programmierung (künstliche Intelligenz). Man spricht hier von »symbolischer Mathematik«.
- Integration einer Funktion, allerdings numerisch, da es für das Bilden der Stammfunktion bekanntlich keine festen Regeln gibt. Aber auch hier verlassen wir angetretene Pfade und bringen ein schnelles und stabiles Integrationsverfahren, das auch bei relativ unstetigen Funktionen und sogar bei Polstellen funktioniert (sofern das uneigentliche Integral existiert)!
- Beim Zeichnen der Funktionsgraphen schließlich werden automatisch die Achsen dem Bildschirm angepaßt. Das kennen Sie vielleicht. Aber es ist auch möglich, Funktionsscharen zu zeichnen, etwa einer Funktion und ihrer 1,2,... Ableitung. Ebenfalls möglich ist die Darstellung einer Funktionsschar in Abhängigkeit von Parametern, wie man es oft bei naturwissenschaftlichen Anwendungen benötigt.

Die Abbildung zeigt den Plot von

$f_a(x)$ = arctan(ax^4 - 8x^2 + 1) für a = 0,8,16,24.

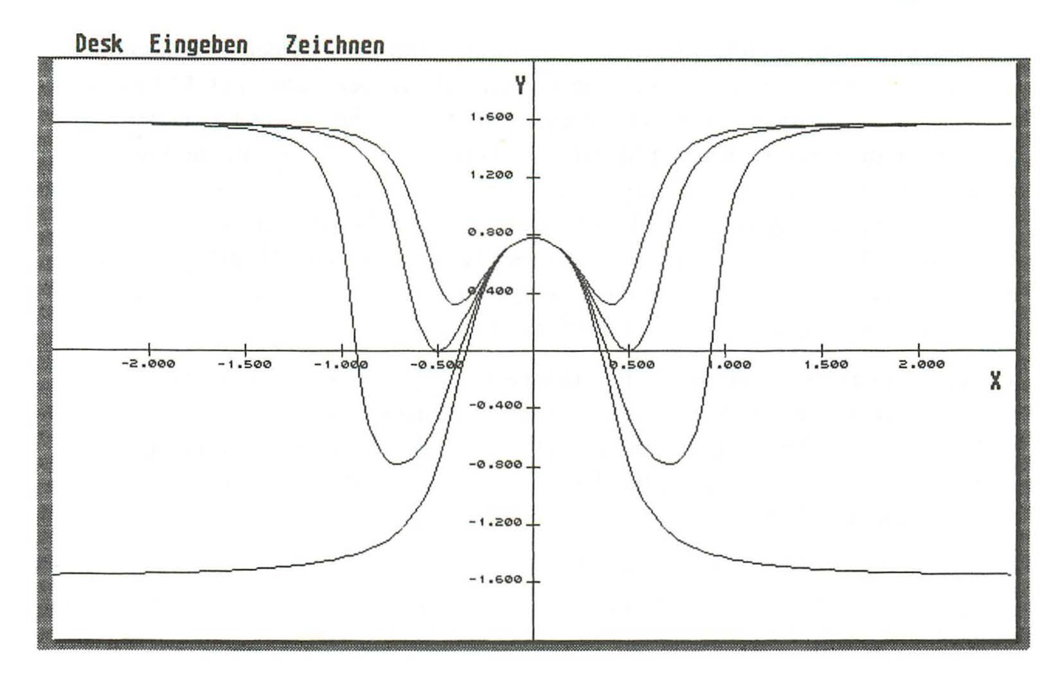


Bild 5.1: Die Funktionenschar f(x,a) = arctan(a x^4-8x + 1) für a=0, 8, 16 und 24

Das Ganze wird trotz der Vielzahl der Features kein unleserliches Programmonster, denn Modula legt es nahe, alles in kleinen Portionen schön, eben modular zu programmieren. In den folgenden Abschnitten besprechen wir Einzelaufgaben und verpacken sie in getrennte Module:

- Parser (mit Scanner)
- Differenzierer
- Optimierer
- Mathelehrer (bringt dem System das Regelwerk bei)
- Integrierer
- Benutzereingaben (»User-Interface«)
- Grafikausgabe

Das Hauptprogramm `ModPlot` (`=MODula-PLOT`) ist dann entsprechend kurz. Bevor Sie weiterlesen, sollten Sie das Programm unbedingt zunächst einmal laufen lassen. Es liegt auf der Diskette in übersetzter Form vor und kann vom Desktop aus gestartet werden.

Interessant ist vielleicht noch, daß die wesentlichen Programmteile wie Parser, Differenzierer und Optimierer auf einem anderen Rechner entwickelt worden ist, was wieder einmal für die

gute Portabilität von Modula spricht. Lediglich die Benutzerschnittstellen wie Funktionseingabe, Ausgabe der Ableitung und des Integrals sind »Atari-spezifisch« mit GEM programmiert. Die Grafik-Ausgabe stützt sich auf unsere Module aus dem Kapitel 4. Im Bereich der Benutzerschnittstellen (Grafik, GEM-Aufrufe) wurden Megamax-spezifische Eigenheiten benutzt. Nur dieser Modul muß also bei Benutzung eines anderen Systems umgeschrieben werden. Sämtliche anderen Teile sind vollkommen portabel, da nur Standardprozeduren benutzt wurden. Aus diesem Grund haben wir auch einen kleinen String-Modul selbst geschrieben. Auch bei den mathematischen Funktionen gehen wir nicht über die gemeinsame Schnittmenge der verschiedenen `MathLib0`-Funktionen hinaus.

Bekanntlich gibt es keine »fertigen« Programme. Es gilt allenfalls der Spruch: »Ein fertiges Programm ist ein veraltetes Programm«. Vielleicht brauchen sie noch Wertetabellen, Nullstellen, Polstellen oder Parameterdarstellung y(t) über x(t), wie bei Lissajous-Figuren. Alles ist durch die Modularisierung schön pflegeleicht, so daß man das Programm nach eigenen Vorstellungen erweitern kann.

Wenngleich unser Beispiel etwas theoriebeladen ist, da die Mathematik nicht jedermanns Hobby ist, so hoffen wir doch, ein Beispiel gefunden zu haben, was viel schönes Modula und etliche »Gags« zeigt.

5.1 Der Modul »Parser«

Wir erläutern die Funktionsweise des Programms am Beispiel der Funktion

$f(x) = x^2 * \sin(x).$

Die Funktion wird zunächst eingegeben. Hierzu erscheint nach Anwahl des Menüpunktes »Funktion eingeben« erscheint die Dialogbox:

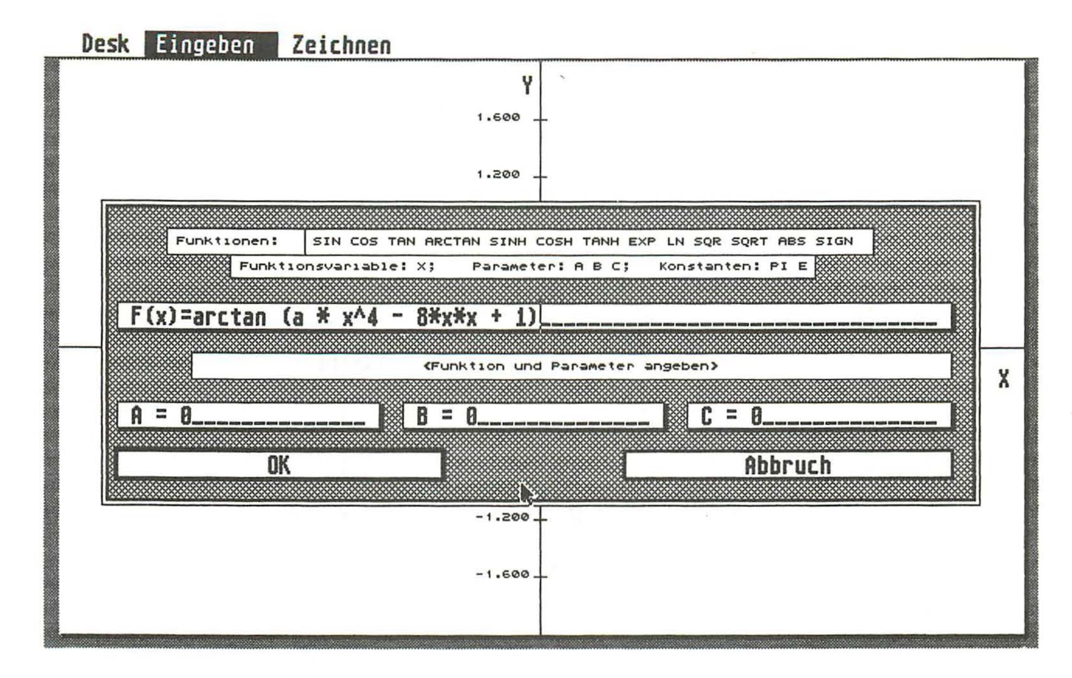


Bild 5.2: Dialogbox zur Funktionseingabe

Die Box listet die implementierten Operatoren und Standardfunktionen auf. Folgende Eingabemöglichkeiten stehen also für die Beispielfunktion zur Verfügung:

`f(x) = x^2*sin x` oder
`f(x) = x * x * sin x` oder
`f(x) = sqr(x)*sin(x)` o. ä.

Wählen wir die erste Eingabemöglichkeit, so wird nach Schließen der Dialogbox – eventuell hat man vorher Definitions- oder Wertebereich geändert – die Prozedur `parse` aufgerufen. Sie erzeugt aus der Zeichenkette den Funktionsbaum auf der folgenden Seite:

Dies besagt folgendes: Der Term "`X^2*SINX`" ist ein Produkt (oberster Operator: »`*`«). Der erste Faktor ist eine Potenz (Operator: »`^`«), die Operanden sind `X` und `2`. Der zweite Faktor besteht aus einer Funktion: `SIN`. Sie hat ein Argument: `X`.

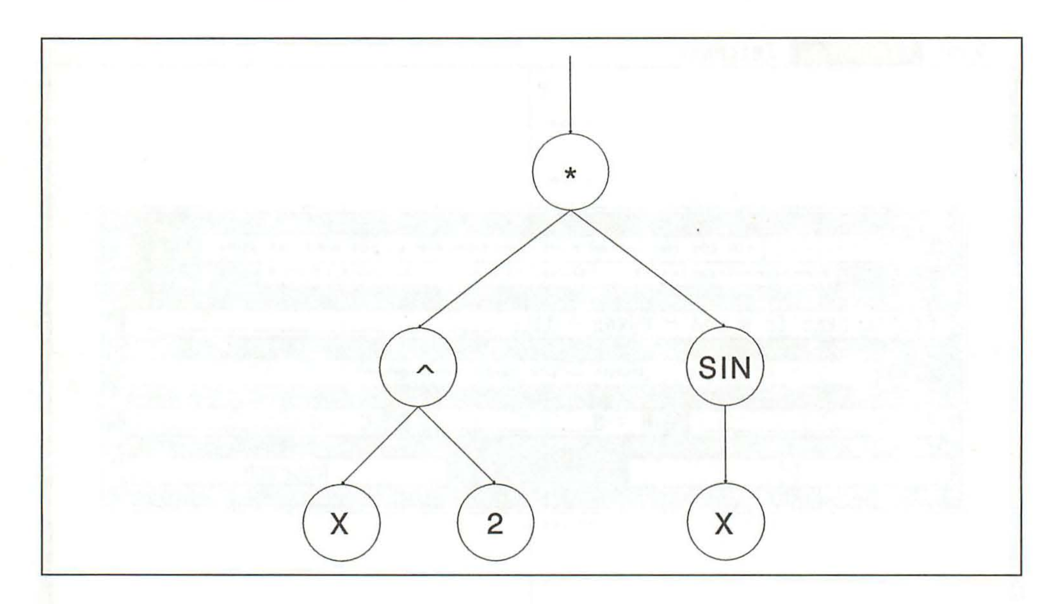


Bild 5.3: Funktionsbaum $x^2 \sin x$

Die Knoten dieses Baumes zeigen auf den varianten Verbund des Typs `ParserNode` (engl. *node*= »Knoten«). Die genaue Struktur geht aus dem Definitionsmodul `Parser` hervor.

```
DEFINITION MODULE Parser;

CONST
    BezHIGH = 16-1;
    MaxHIGH = 255;
    TYPE      RR = REAL;
    INDEX = CARDINAL;
    BezString = ARRAY[0..BezHIGH] OF CHAR;
    MaxString = ARRAY[0..MaxHIGH] OF CHAR;

    FunRl = PROCEDURE(RR): RR;          (* Reelle, eindimensionale Funktion *)

    ErrorType = (
        errOK,            (* Kein Fehler *)
        errCharacter,     (* unerlaubtes Zeichen im Text *)
        errBezeichner,    (* undeklarierter Bezeichner *)
        errKlammerAuf,    (* "(" erwartet *)
        errKlammerZu,     (* ")" erwartet *)
        errKomma,         (* "," erwartet *)
        errAusdruck,      (* Arithmetischer Ausdruck erwartet *)
```

```
        errOperator    (* Operator erwartet *)
        );

    SymType = (
        SymUnbekannt, SymEnde,
        KlammerAuf, KlammerZu,
        OpMinus, OpPlus, OpDurch, OpMal, OpHoch, OpNeg,
        SymFunRl, SymVarRR, SymKoRR);

    BezPtr = POINTER TO Bezeichner;
    Bezeichner = RECORD
        next: BezPtr;
        Name: BezString;
        CASE BezArt: SymType OF
            SymFunRl : Fktl    : FunRl |
            SymVarRR : ValueRR : RR |
        END END;

    ParserPtr = POINTER TO ParserNode;
    ParserNode = RECORD
        CASE OperArt: SymType OF
            OpNeg:
                Operand: ParserPtr |
            OpMinus, OpPlus, OpDurch, OpMal, OpHoch:
                Operand1, Operand2: ParserPtr |
            SymFunRl:
                BezFktl: BezPtr;
                Parameter: ParserPtr |
            SymVarRR:
                Variable: BezPtr |
            SymKoRR:
                KoRR: RR
        END END;

VAR
    ScanZeile: MaxString;
    ScanPosition, ScanErrPos: CARDINAL;
    SyntaxError, ArithmeticError: BOOLEAN;
    ErrorArt: ErrorType;
PROCEDURE HoleBezeichner(name: BezString): BezPtr;
PROCEDURE LerneFunktion(name: BezString; Funktion: FunRl);
PROCEDURE LerneVariable(name: BezString; VarWert: RR);
PROCEDURE SetzeVariable(bez: BezPtr; wert: RR);
PROCEDURE parse(zeile: ARRAY OF CHAR): ParserPtr;
PROCEDURE BaumZuString(baum: ParserPtr; VAR formel: ARRAY OF CHAR);
```

```
PROCEDURE LoescheBaum(VAR baum: ParserPtr);
PROCEDURE NewNode(VAR node: ParserPtr);
PROCEDURE berechne(baum: ParserPtr; xWert: RR): RR;
PROCEDURE PrinTree(p: ParserPtr);
END Parser.
```

Der Parser hat also die Aufgabe, den Funktionsterm als Zeichenkette »grammatikalisch« zu analysieren und – falls diese fehlerfrei ist – den entsprechenden Baum aufzubauen. Ansonsten wird die Fehlerposition zurückgegeben. Das Wort »Parser« leitet sich vom engl. *to parse* ab, was soviel heißt wie »zerteilen«. Gemeint ist »einen Satz nach seiner grammatikalischen Struktur zerlegen«.

Der Modul `Parser` ist der längste in diesem Programm. Seine Struktur gliedert sich wie folgt:

1. Importliste.
2. Lokaler Modul `Scanner`, der die Zeichenkette zeichenweise abgeht und die entsprechenden Symbole erzeugt.
3. Eigentlicher Parser; er baut den »Parserbaum« auf.
4. Prozedur `Berechne`, die zu jedem übergebenen x-Wert aus dem Parserbaum den Funktionswert f(x) errechnet.
5. Initialisierungsteil des Moduls `Parser`.

Wir sehen hier die grobe Struktur wieder, die jedem Compiler zugrundeliegt: Scanner-Parser-Codegenerator. So funktioniert auch Ihr Modula-Compiler! Der Code-Erzeuger wandelt nämlich die vom Parser »vorverdauten« Textbausteine in 68000er-Code um, der Scanner hat den Text für den Parser »vorgekaut«. Er trennt die einzelnen Wörter (Bezeichner, Operatoren) voneinander und ordnet ihnen symbolische Namen zu. Der einzige Unterschied zu unserem Vorgehen besteht darin, daß kein echter Maschinencode erzeugt wird. Vielmehr wird aus dem Parserbaum mit einer Wertbelegung für die Variable `x` der Funktionswert `f(x)` in der Hochsprache errechnet. Es handelt sich also mehr um ein »interpretieren« des Parser-Baumes.

Das MSM2-System liefert bereits eine fertige Prozedur `Fctcomp` im Modul `Formelcompiler`, die es gestattet, einen Funktionsstring sofort in Maschinencode zu übersetzen. Dies ist sehr komfortabel. Man benötigt den gesamten Modul `Parser` nicht, wenn man nur Funktionswerte ausrechnen will. Zudem ist die Berechnung sehr schnell, da echter Maschinencode erzeugt wird. Der Code wird zur Laufzeit erzeugt und auf dem Heap abgelegt. Bei einer Wertermittlung `f(x)` wird dieser Code einfach angesprungen und mit `x` abgearbeitet.

Wenn sie also diesen Modula-Compiler besitzen, können Sie sehr schnell selbst ein eigenes Funktionenprogramm schreiben, wenn sie sich dabei auf die folgenden Themen beschränken:

- Erstellung einer Wertetabelle
- Zeichnen von Funktionsgraphen
- Ermittlung von Nullstellen, Minima und Maxima
- numerisches Ableiten und Integrieren

Unser Weg ist zwar sehr viel steiniger, hat aber zwei Vorteile:

- Der gezeigte Parser ist auf allen Modula-Systemen einsetzbar. Da bei der Berechnung der Funktionswerte lediglich ein paar Zeiger übergeben werden, arbeitet es auch sehr rasant.
- Der Parser-Baum gestattet auf einfache Weise die algebraische Ermittlung der Ableitung. Diese kann als Zeichenkette ausgegeben werden, was sehr viel leistungsstärker ist als das allgemein übliche Verfahren der numerischen Ableitung durch Annäherung durch Tangentensteigung.

Nun gehen wir auf die Funktionsweise des Moduls ein. Der eingegebene Funktionsstring, der für den Benutzer lesbar ist, muß also in einen Funktionsbaum (Parserbaum) umgewandelt werden, der für den Rechner handlich ist. Diese Aufgabe teilen sich im wesentlichen zwei Funktionseinheiten: der Parser und der Scanner. Der Scanner sucht die eingegebene Zeichenkette nach zusammengehörenden Wörtern (Zahlen, Variablen- oder Funktionsnamen, Operationszeichen...) ab und liefert sie dem Parser in Form von Symbolen. Dabei werden Konstanten (die als Zeichenfolge vorliegen) in REAL-Zahlen umgewandelt. Erkennt der Scanner einen Bezeichner (Kennzeichen: beginnt mit einem Buchstaben), sieht er über die Funktion `HoleBezeichner` in einer Liste (der `BezeichnerListe`) nach, ob der Bezeichner definiert ist.

Der Parser wird über die Funktion `parse` mit dem Funktionsstring aufgerufen. `parse` übergibt mit `StarteScanner` die Zeichenkette an den Scanner, damit er sie sich global in der Variablen `ScanZeile` merken kann. Um Konflikte mit Groß- und Kleinschreibung zu vermeiden, wird der String zunächst einmal mit `CopyCap` kopiert und dabei in Großbuchstaben umgewandelt. Benötigt der Parser ein neues Symbol vom Scanner, ruft er die Funktion `LiesSymbol` auf. Anschießend findet er in dem Verbund `ScanNode` das nächste Symbol (vom Typ: `SymType`).

Ein Term ist nicht einfach eine Reihung von Zahlen und Operatoren, sondern besitzt eine gewisse Struktur. Es gibt Operatoren, die einen Vorrang vor anderen haben (»*« bindet stärker als »+«). Zuerst müssen die Zahlen, die mit »^« verbunden sind, zuammengefaßt werden (»hoch« hat die höchste Priorität). Diese »Faktoren« werden durch »*« und »/« zu »Produkten« zusammengefaßt und diese wieder mit »+« und »-« (niedrigste Priorität) zu einer Summe. Auf der ersten Ebene haben wir also Plus und Minus. Das Vorzeichen und die Funktionen haben den stärksten Vorrang (auf der letzten Ebene). Zusätzlich bilden die Klammern weitere Gruppierungen. Das ganze läßt sich als Syntaxdiagramm darstellen:

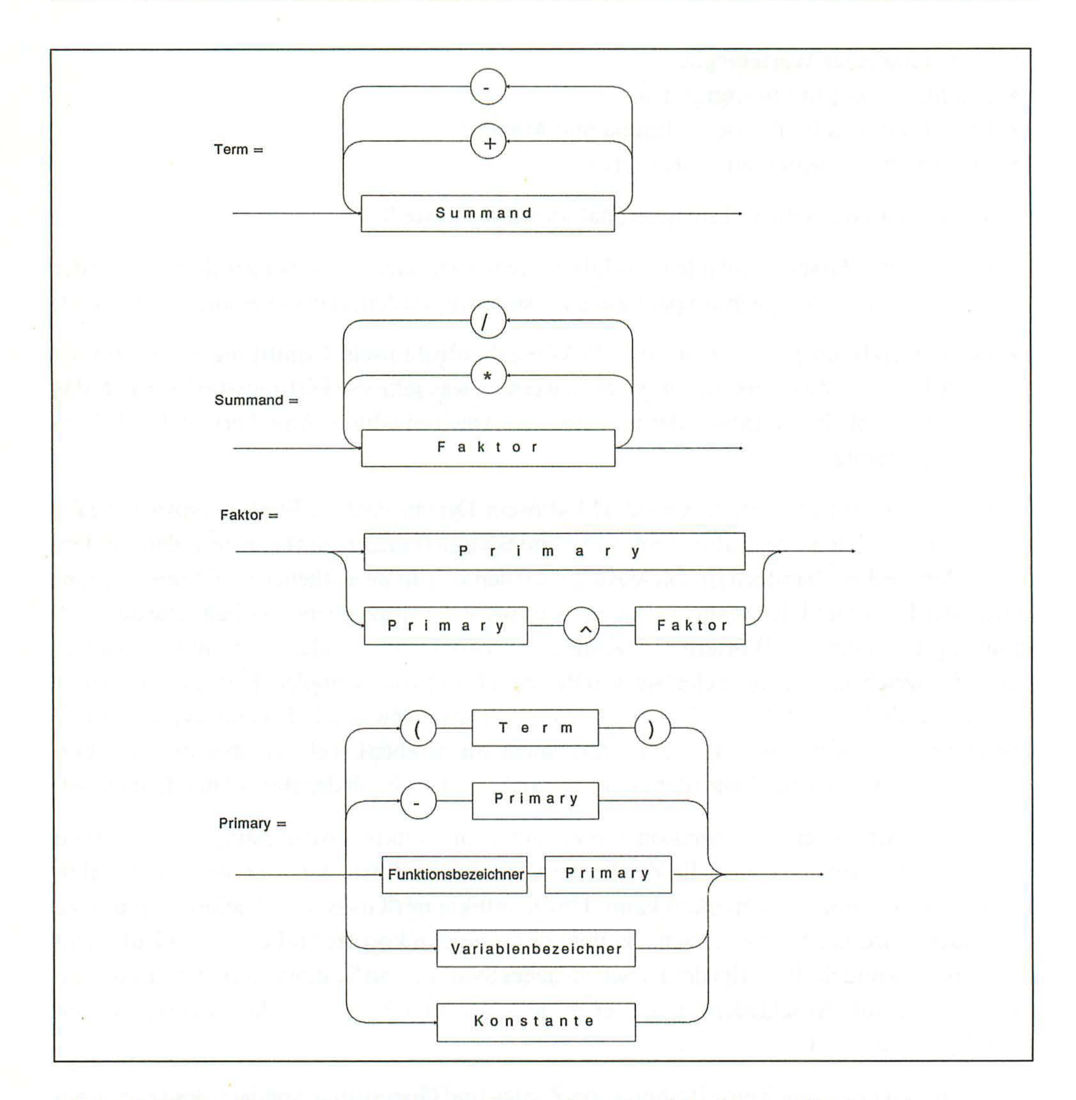


Bild 5.4: Syntaxdiagramme des Parsers (Term, Summand, Faktor und Primary)

Der Parser arbeitet vollkommen analog zu diesen Diagrammen. Jedes der vier Syntaxdiagramme entspricht einer Prozedur `Term`, `Summand`, `Faktor` und `Primary`. Ein eckiger Kasten wird dabei in einen Aufruf der entsprechenden Funktion umgesetzt, ein runder in einen Aufruf des Scanners, der das jeweils nächste Symbol des Funktionsterms liefert. Ein vom Scanner geliefertes Symbol ist also für den Parser ein Terminalsymbol (vgl. Kapitel 1.1.3). Die Syntaxdiagramme für `Funktionsbezeichner`, `Variablenbezeichner` und `Konstante` können zunächst noch undefiniert bleiben. Der Parser arbeitet abstrakt mit ihnen; im Gesamtsystem werden sie an späterer Stelle festgelegt.

Betrachten wir das Syntaxdiagramm »Term«. Ein »Term« besteht aus einem Summanden, dem beliebig oft »+« bzw. »-« ein weiterer Summand folgen kann. Für die Funktion `Term` bedeutet das folgendes:

1. Summand einlesen (mit der Funktion `Summand`).

2. Solange ein »+« oder »-«-Operator folgt, diesen Operator einlesen und einen weiteren Summanden lesen.

Gleichzeitig muß natürlich der Parserbaum aufgebaut werden. Im einzelnen sieht das so aus:

1. `Term` ruft `Summand` auf und erhält dabei einen Teilbaum für einen »Summanden«. `Summand` hat dabei aus dem Funktions-String einen kompletten »Summanden« gelesen.

2. Nach dem Syntaxdiagramm kann nach einem Summanden ein »+« oder »-« folgen. Ist das nicht der Fall, besteht die »Summe« nur aus dem einzigen Summanden, den `Term` von `Summand` als Teilbaum erhalten hat. Diesen kann diese Funktion dann »Term« zurückgeben.

3. Im anderen Fall liegt eine Summe oder Differenz vor, und `Term` muß dazu einen entsprechenden Baum liefern. Dazu ruft `Term` jetzt `BaueSymbol` auf: diese Prozedur liest dazu den nächsten Operator ein (das »+« bzw. »-«) und erzeugt einen entsprechenden Knoten.

4. Dieser Knoten benötigt zwei Teilbäume (`Operand1` und `Operand2`) als Operanden. Den ersten hat `Term` zuvor von `Summand` erhalten. Den zweiten erhält `Term` durch einen weiteren Aufruf von `Summand`.

5. Wenn dann kein weiteres »+« oder »-« folgt, kann dieser Knoten zurückgegeben werden. Ansonsten muß ein weiterer Summand angehängt werden. Dazu geht es weiter wie in 3.

Wir sehen: `Term` liest einen kompletten »Term« ein und liefert einen Baum. Ebenso verfährt die Prozedur `Summand`; sie beschränkt sich im wesentlichen darauf, die Prozedur der nächsten Ebene aufzurufen: `Faktor`.

Die Aufrufkette landet auf der untersten Ebene bei der Prozedur `Primary`. Sie liest eine kleinstmögliche Einheit – wie zum Beispiel eine Konstante oder Variable – ein. Leider fiel uns hiefür kein geeigneter deutscher Bezeichner ein. Die verzweigte Struktur des Syntaxdiagrammes spiegelt sich in der CASE-Anweisung wieder. Wenn `Primary` eine öffnende Klammer »(« entdeckt, kann wieder ein kompletter Term folgen und es wird wieder `Term` aufgerufen. Die Struktur ist also eine Rekursion »im Kreis«, an der vier Prozeduren beteiligt sind:

```
Term-->Summand-->Faktor-->Primary-->Term...
```

Wir sehen folgendes: Variablen, Konstanten, Funktionen und Klammern werden von der Prozedur `Primary` gelesen. `Primary` erzeugt dazu ein entsprechendes »Blatt« des Baumes.

Operatoren werden von einer Prozedur der entsprechenden Ebene gelesen. Es wird ein neuer Knoten erzeugt, dessen Teilbäume von Funktionen der niedrigeren Ebene geliefert werden.

Dieses Beispiel reicht zum Verständnis des Parsers. Wenn sie Lust haben, können Sie den folgenden Implementationsmodul durchgehen und es nacharbeiten. Beim Durchlesen des Quelltextes empfiehlt sich die oben genannte Gliederung 1.-5. des Parsers im »Hinterkopf« zu behalten, dann wird es kein Problem mehr sein.

```
IMPLEMENTATIONMODULE Parser;

FROM SYSTEM    IMPORT TSIZE;
FROM Storage   IMPORT ALLOCATE, DEALLOCATE;
FROM MathLib0  IMPORT power;
FROM ZKetten   IMPORT gleich;
IMPORT ZKetten;

(*==================================================================== *)
(*==================== Zeichen-Behandlung=========================== *)

MODULE StringModul;

EXPORT Ziffer, ReelleZiffer, Buchstabe, AlphaNum;

PROCEDURE Ziffer(z: CHAR): BOOLEAN;
BEGIN
    RETURN ("0" <= z) AND (z <= "9")
END Ziffer;

PROCEDURE Buchstabe(z: CHAR): BOOLEAN;
BEGIN
    RETURN ("A" <= z) AND (z <= "Z")
END Buchstabe;

PROCEDURE AlphaNum(z: CHAR): BOOLEAN;
BEGIN
    RETURN Ziffer(z) OR Buchstabe(z);
END AlphaNum;

END StringModul;

(* ================================================================== *)
```

```
PROCEDURE error(e: ErrorType);
BEGIN
    IF e = errOK THEN           SyntaxError := FALSE;
        ErrorArt := errOK
    ELSE
        IF NOT SyntaxError THEN
            ErrorArt := e;
            SyntaxError := TRUE END
    END
END error;

(* ------------------------------------------------------------------------ *)

VAR
    BezeichnerListe: BezPtr;        (* Liste aller Bezeichner (Funktionen) *)

PROCEDURE HoleBezeichner(name: BezString): BezPtr;
( * liefert zu einem Bezeichner-Namen einen Pointer auf den Bezeichner-
  * deskriptor, wenn er in der Bezeichnerliste steht, sonst NIL  *)
VAR p: BezPtr;
BEGIN
    p := BezeichnerListe;
    WHILE (p <> NIL) AND NOT gleich(p^.Name, name) DO p := p^.next END;
    RETURN p
END HoleBezeichner;

PROCEDURE LerneFunktion(
        name: BezString;
        funk: FunRl);
VAR
    neuer: BezPtr;
BEGIN
    IF HoleBezeichner(name) <> NIL THEN HALT END;       (* schon definiert! *)
    ALLOCATE(neuer, TSIZE(Bezeichner));                  (* Platz holen ... *)
    neuer^.next := BezeichnerListe;                      (* fuer Verkettung *)
    neuer^.Name := name;                                      (* belegen... *)
    neuer^.BezArt := SymFunRl;
    neuer^.Fktl := funk;
    BezeichnerListe := neuer                        (* in die Liste einbauen *)
END LerneFunktion;
```

```
PROCEDURE LerneVariable(
        name: BezString;
        wert: RR);
VAR
    neuer: BezPtr;
BEGIN
    IF HoleBezeichner(name) <> NIL THEN HALT END;          (* schon definiert! *)
    ALLOCATE(neuer, TSIZE(Bezeichner));                      (* Platz holen... *)
    neuer^.next := BezeichnerListe;                             (* Belegen ... *)
    neuer^.Name := name;
    neuer^.BezArt := SymVarRR;
    neuer^.ValueRR := wert;
    BezeichnerListe := neuer                      (* ... in die Liste einbauen *)
END LerneVariable;

PROCEDURE SetzeVariable(bez: BezPtr; wert: RR);
BEGIN
    IF bez = NIL THEN HALT END;
    IF bez^.BezArt <> SymVarRR THEN HALT END;            (* muss Variable sein *)
    bez^.ValueRR := wert
END SetzeVariable;

PROCEDURE ClearNode(VAR node: ParserNode);
(* loescht einen ParserKnoten. Rein prophylaktische Angelegenheit *)
BEGIN
    WITH node DO
        CASE OperArt OF
            OpMinus, OpPlus, OpMal, OpDurch, OpHoch:
                Operand1 := NIL;
                Operand2 := NIL |
            OpNeg:
                Operand := NIL |
            SymFunR1:
                Parameter := NIL |
            SymVarRR:
                Variable := NIL |
        ELSE
            (* NICHTS, nix zum loeschen da *)
        END
    END
END ClearNode;
```

```
PROCEDURE BaumZuString(baum: ParserPtr; VAR formel: ARRAY OF CHAR);
VAR
    s: ZKetten.StrRR;
    pos: INDEX;
    ueberlauf: BOOLEAN;

    PROCEDURE schreib(s: ARRAY OF CHAR);
    VAR i: INDEX;
    BEGIN
        i := 0;
        WHILE NOT ueberlauf AND (i <= HIGH(s)) AND (s[i] <> 0C) DO
            IF pos <= HIGH(formel) THEN
                formel[pos] := s[i];
                INC(pos); INC(i)
            ELSE
                ueberlauf := TRUE
            END
        END
    END schreib;

    PROCEDURE wandle(p: ParserPtr; level: SymType);
    BEGIN WITH p^ DO
        IF OperArt < level THEN schreib("(") END;
        CASE OperArt OF
            OpNeg:
                schreib("- ");
                wandle(Operand, OpNeg) |
            OpMinus:
                wandle(Operand1,OpMinus);
                schreib(" - ");
                wandle(Operand2,OpDurch) |
            OpPlus:
                wandle(Operand1, OpPlus);
                schreib(" + ");
                wandle(Operand2, OpDurch) |
            OpDurch:
                wandle(Operand1, OpDurch);
                schreib(" / ");
                wandle(Operand2, OpHoch) |
            OpMal:
                wandle(Operand1, OpMal);
```

```
                schreib(" * ");
                wandle(Operand2, OpHoch) |
            OpHoch:
                wandle(Operand1, OpNeg);
                schreib(" ^ ");
                wandle(Operand2, OpHoch) |
            SymVarRR:
                schreib(Variable^.Name) |
            SymFunR1:
                schreib(BezFkt1^.Name);
                schreib(" ");
                wandle(Parameter, OpNeg) |
            SymKoRR:
                ZKetten.RzuS(KoRR,3,s); schreib(s)
        END;
        IF OperArt < level THEN schreib(")") END;
    END END wandle;

BEGIN
    pos := 0;
    ueberlauf := FALSE;
    wandle(baum, SymEnde);
    IF pos <= HIGH(formel) THEN formel[pos] := 0C END;
    IF ueberlauf THEN formel[HIGH(formel)] := "#" END
END BaumZuString;

(* ---------------------------------------------------------------
:: nur zu Testzwecken: Ausgabe des kompletten Parser-Baumes
*)

PROCEDURE PrinTree(baum: ParserPtr);
VAR i,tiefe: CARDINAL;
(* -------- Wird ausgeklammert, da er Ausgaben macht! ---------- >

PROCEDURE printree1 (p: ParserPtr);
BEGIN
    WriteLn;
    FOR i := 1 TO tiefe DO WriteString("..|") END;
    WriteString("--");
    IF p = NIL THEN
        WriteString("*** NIL ***")
    ELSE
        INC(tiefe);
```

```
        CASE p^.OperArt OF
            OpNeg:
                WriteString("(- (Neg))");
                printreel(p^.Operand) |
            OpPlus, OpMinus, OpMal, OpDurch, OpHoch:
                CASE p^.OperArt OF
                    OpPlus: WriteString("[+]") |
                    OpMinus: WriteString("[-]") |
                    OpMal: WriteString("[*]") |
                    OpDurch: WriteString("[/]") |
                    OpHoch: WriteString("[HOCH]")
                END;
                printreel(p^.Operand1);
                printreel(p^.Operand2) |
            SymFunR1:
                WriteString(p^.BezFktl^.Name); WriteString("(.)");
                printreel(p^.Parameter) |
            SymVarRR:
                WriteString(p^.Variable^.Name); WriteString(" = ");
                WriteReal(p^.Variable^.ValueRR, 20,11) |
            SymKoRR:
                WriteReal(p^.KoRR, 20,11)
        END;
        DEC(tiefe)
    END
END printreel;

BEGIN
    tiefe := 0;
    printreel(baum)
 <------------------------------------ Ende der Ausklammerung *)
END PrinTree;

(* ==================================================================
 * =================  1. Teil: der Scanner ====================== *)

MODULE Scanner;

IMPORT
    gleich, Ziffer, ReelleZiffer, Buchstabe, AlphaNum,
    MaxHIGH, ScanZeile, ScanErrPos, ScanPosition,
    ParserNode, SymType, BezString, BezPtr, HoleBezeichner,
    INDEX, RR, ClearNode,
    error, ErrorType, SyntaxError;
```

```
FROM ZKetten IMPORT CopyCAP, reell;

EXPORT
    StarteScanner, LiesSymbol, ScanNode;

VAR
    ScanNode: ParserNode;                    (* naechste Symbol vom Scanner  *)
    ScanChar: CHAR;

PROCEDURE LiesZeichen(): CHAR;
VAR z: CHAR;
BEGIN
    z := ScanChar;
    IF (ScanPosition <= MaxHIGH) AND (ScanZeile[ScanPosition] <>0C) THEN
        INC(ScanPosition);
        ScanChar := ScanZeile[ScanPosition]
    ELSE
        ScanChar := 0C;                                      (* Zeile zu Ende *)
    END;
    RETURN z
END LiesZeichen;

PROCEDURE SkipZeichen;
VAR dummy: CHAR;
BEGIN
    dummy := LiesZeichen();
END SkipZeichen;

PROCEDURE LiesWort(VAR s: ARRAY OF CHAR);
(* Liest genau ein Wort aus der 'ScanZeile' nach 's' *)
VAR
    i: INDEX;
    z: CHAR;
BEGIN
    i := 0;
    REPEAT
        z := LiesZeichen();
        IF i <= HIGH(s) THEN s[i] := z; INC(i) END
    UNTIL NOT AlphaNum(ScanChar);
    IF i <= HIGH(s) THEN s[i] := 0C END
```

```
END LiesWort;

PROCEDURE LiesZahl;
(* einlesen einer RR- Konstante *)
VAR
    r,d10: RR;
BEGIN
    r := 0.0;
    WHILE Ziffer(ScanChar) DO r := 10.0 * r +
          FLOAT( ORD(LiesZeichen()) - ORD("0") ) END;
    IF ScanChar = "." THEN            (* es kommen noch Nachkommastellen... *)
        SkipZeichen;
        d10 := 0.1;
        WHILE Ziffer(ScanChar) DO
            r := r + FLOAT(ORD(LiesZeichen())-ORD("0")) * d10;
            d10 := 0.1 * d10 END
    END;
    ScanNode.OperArt := SymKoRR;
    ScanNode.KoRR := r
END LiesZahl;

PROCEDURE LiesSymbol;
(*
 * Liest aus der EingabeZeile ein Symbol und traegt es in 'ScanNode' ein
 *)
VAR
    BezName: BezString;
    bezei: BezPtr;
BEGIN
    IF SyntaxError THEN HALT END;           (* Parser nicht richtig gestoppt *)
    WHILE ScanChar = " " DO SkipZeichen END;
    ScanErrPos := ScanPosition;    (* Position merken, falls Fehler auftritt *)
    IF Buchstabe(ScanChar) THEN              (* -------- Bezeichner -------- *)
        LiesWort(BezName);
        bezei := HoleBezeichner(BezName);
        IF bezei = NIL THEN
            ScanNode.OperArt := SymUnbekannt;
            error(errBezeichner)
        ELSE
            ScanNode.OperArt := bezei^.BezArt
        END;
        ClearNode(ScanNode);
```

```
        CASE ScanNode.OperArt OF
            SymFunRl:
                ScanNode.BezFktl := bezei |
            SymVarRR:
                ScanNode.Variable := bezei |
            SymUnbekannt: (* nix *)
        END
    ELSIF Ziffer(ScanChar) THEN             (* -------- Konstante ----------- *)
        LiesZahl
    ELSE                        (* ---- sonst: Operator oder Satzzeichen ---- *)
        CASE ScanChar OF
            "(":    ScanNode.OperArt := KlammerAuf |
            ")":    ScanNode.OperArt := KlammerZu |
            "-":    ScanNode.OperArt := OpMinus |
            "+":    ScanNode.OperArt := OpPlus |
            "/":    ScanNode.OperArt := OpDurch |
            "*":    ScanNode.OperArt := OpMal |
            "^":    ScanNode.OperArt := OpHoch |
            0C:     ScanNode.OperArt := SymEnde                (* Zeile zu Ende *)
        ELSE
            ScanNode.OperArt := SymUnbekannt;
            error(errCharacter)
        END;
        SkipZeichen;
        ClearNode(ScanNode)               (* sicherheitshalber durchloeschen *)
    END;
END LiesSymbol;

PROCEDURE StarteScanner(VAR eingabe: ARRAY OF CHAR);
(* ---  Initialisiert den Scanner  --- *)
BEGIN  (* Reihenfolge Wichtig! *)
    CopyCAP(eingabe, ScanZeile);                        (* Eingabe  merken *)
    ScanErrPos := 0;                                    (* Startpositionen *)
    ScanPosition := 0;
    ScanChar := ScanZeile[0];                      (* Erstes Zeichen Holen *)
    error(errOK);                                  (* Fehler zuruecksetzen *)
    LiesSymbol;                                   (* erstes Symbol einlesen *)
END StarteScanner;

END Scanner;

(* ---------------------------------------------------------------------- *)

PROCEDURE NewNode(VAR node: ParserPtr);
```

```
(* -----   Erzeugt einen neuen ParserNode   ----- *)
BEGIN
    ALLOCATE(node, TSIZE(ParserNode));
    IF node = NIL THEN HALT END;              (* kein Speicher: brutal raus! *)
    node^.OperArt := SymUnbekannt;                      (* sicherheitshalber *)
END NewNode;

PROCEDURE LoescheBaum(VAR p: ParserPtr);
(* Loescht einen ParserBaum (gibt damit den Speicher wieder frei) *)
BEGIN
    IF p <> NIL THEN
        CASE p^.OperArt OF
            OpNeg:
                LoescheBaum(p^.Operand) |
            OpMinus, OpPlus, OpMal, OpDurch, OpHoch:
                LoescheBaum(p^.Operand1);
                LoescheBaum(p^.Operand2) |
            SymFunR1:
                LoescheBaum(p^.Parameter) |
            SymVarRR, SymKoRR:
                (* Nix, gibt keine UnterBaeume *)
            ELSE HALT                         (* da war wohl der Baum kaputt... *)
        END;
        DEALLOCATE(p, TSIZE(ParserNode));
        p := NIL
    END
END LoescheBaum;

PROCEDURE SkipSymbol;
BEGIN
    IF NOT SyntaxError THEN LiesSymbol END
END SkipSymbol;

PROCEDURE BaueSymbol(VAR nodePtr: ParserPtr);
BEGIN
    IF SyntaxError THEN HALT END;            (* Parser nicht richtig gestoppt *)
    NewNode(nodePtr);
    nodePtr^ := ScanNode;
    LiesSymbol
END BaueSymbol;

PROCEDURE MussSymbol(ErwartetesSymbol: SymType; err: ErrorType);
BEGIN
    IF ScanNode.OperArt = ErwartetesSymbol
```

```
        THEN SkipSymbol
        ELSE error(err)
    END
END MussSymbol;

(* ==================================================================
 * =================  2. der eigentliche Parser ====================
 *)

FORWARD makeTerm(): ParserPtr;

PROCEDURE makePrimary(): ParserPtr;
VAR
    node: ParserPtr;
BEGIN
    IF SyntaxError THEN RETURN NIL END;
    CASE ScanNode.OperArt OF
        KlammerAuf:                           (* ---- "(" <Term> ")" ---- *)
            SkipSymbol;
            node := makeTerm();
            MussSymbol(KlammerZu, errKlammerZu) |
        OpMinus:                              (* ---- "-" <Primary> ---- *)
            BaueSymbol(node);
            node^.OperArt := OpNeg;           (* das "-" hier monadisch *)
            node^.Operand := makePrimary() |
        SymFunR1:                    (* ---- <Bezeichner> <Primary> ---- *)
            BaueSymbol(node);
            node^.Parameter := makePrimary() |
        SymVarRR, SymKoRR:             (* ---- Variable / Konstante ---- *)
            BaueSymbol(node)
    ELSE
        error(errAusdruck);
        RETURN NIL
    END;
    RETURN node;
END makePrimary;

PROCEDURE makeFaktor(): ParserPtr;
VAR node, p: ParserPtr;
BEGIN
    IF SyntaxError THEN RETURN NIL END;
    node := makePrimary();
    IF ScanNode.OperArt = OpHoch THEN   (* ---- <Primary> "^" <Faktor> ---- *)
```

```
        BaueSymbol(p);
        p^.Operand1 := node;
        p^.Operand2 := makeFaktor();
        node := p END;
    RETURN node;
END makeFaktor;

PROCEDURE makeSummand(): ParserPtr;
VAR
    node, p: ParserPtr;
BEGIN
    IF SyntaxError THEN RETURN NIL END;
    node := makeFaktor();
    LOOP
        IF SyntaxError THEN EXIT END;
        CASE ScanNode.OperArt OF
            OpMal, OpDurch:
                BaueSymbol(p);
                p^.Operand1 := node;
                p^.Operand2 := makeFaktor();
                node := p
        ELSE EXIT
        END
    END;
    RETURN node;
END makeSummand;

PROCEDURE makeTerm(): ParserPtr;
VAR
    node,p: ParserPtr;
BEGIN
    IF SyntaxError THEN RETURN NIL END;
    node := makeSummand();
    LOOP
        IF SyntaxError THEN EXIT END;
        CASE ScanNode.OperArt OF
            OpPlus, OpMinus:
                BaueSymbol(p);
                p^.Operand1 := node;
                p^.Operand2 := makeSummand();
                node := p
        ELSE EXIT;
```

```
        END
    END;
    RETURN node;
END makeTerm;

PROCEDURE parse(EingabeZeile: ARRAY OF CHAR): ParserPtr;
VAR node: ParserPtr;

BEGIN
    StarteScanner(EingabeZeile);                    (* Scanner initialisieren *)
    node := makeTerm();                                 (* Einen Term abholen *)
    PrinTree(node);
    IF ScanNode.OperArt <> SymEnde THEN
        error(errOperator);
        LoescheBaum(node) END;               (* Fehler ==> Baum wieder abbauen *)
    RETURN node
END parse;

(* ===================================================================
 * ==============   3. der Rechner (berechnet den Baum) =============
 *)

VAR VarX: BezPtr;

PROCEDURE eval(p: ParserPtr): RR;
CONST
    schrott = 888.888;
VAR
    temp: RR;
BEGIN
    IF p = NIL THEN HALT END;       (* Da ist wohl der Baum nicht vollstaendig *)
    IF ArithmeticError THEN RETURN schrott END;      (* Bei Fehler: Ungueltig *)
    CASE p^.OperArt OF
        OpNeg:
            RETURN - eval(p^.Operand) |
        OpPlus:
            RETURN eval(p^.Operand1) + eval(p^.Operand2) |
        OpMinus:
            RETURN eval(p^.Operand1) - eval(p^.Operand2) |
        OpMal:
            RETURN eval(p^.Operand1) * eval(p^.Operand2) |
        OpDurch:
            temp := eval(p^.Operand2);         (* Divisor zuerst berechnen ... *)
            IF temp = 0.0 THEN                 (* Division durch Null abfangen *)
```

```
                ArithmeticError := TRUE;                          RETURN schrott END;
            RETURN eval(p^.Operand1) / temp |
        OpHoch:
            temp := eval(p^.Operand1);
            IF temp <= 0.0 THEN                           (* negative Basis abfangen * )
                ArithmeticError := TRUE;
                RETURN schrott END;
            RETURN power(temp,eval(p^.Operand2)) |
        SymFunR1:
            RETURN p^.BezFkt1^.Fkt1( eval(p^.Parameter) ) |
        SymVarRR:
            RETURN p^.Variable^.ValueRR |
        SymKoRR:
            RETURN p^.KoRR
        ELSE HALT               (* Baum defekt *)
    END
END eval;

PROCEDURE berechne(baum: ParserPtr; xWert: RR): RR;
BEGIN
    SetzeVariable(VarX, xWert);
    ArithmeticError := FALSE;   (* noch ist alles OK... *)
    RETURN eval(baum)
END berechne;

(* ==================================================================
 * ==================    Modul-Initialisierung =====================
 *)

BEGIN
    BezeichnerListe := NIL;
    LerneVariable("X",0.0);
    VarX := HoleBezeichner("X");
END Parser.
```

Der Parser und die Benutzerschnittstelle brauchen einige Stringprozeduren. Wir gehen hier einmal nicht den üblichen Weg des Imports aus Systemmodulen, sondern schreiben eigene, auf unsere Zwecke zielende Routinen. Vielleicht ist es für den Leser ganz interessant, wie man so etwas macht. Natürlich sind sie in einem eigenen externen Modul verpackt:

```
DEFINITION MODULE ZKetten;

TYPE
    RR    = REAL;                     (* REAL oder LONGREAL *)
    StrRR = ARRAY[0..15] OF CHAR;
    INDEX = CARDINAL;      (* Bei manchen Compilern: INTEGER *)

PROCEDURE gleich(VAR sl,s2: ARRAY OF CHAR): BOOLEAN;
PROCEDURE CopyCAP(VAR quelle, ziel: ARRAY OF CHAR);

PROCEDURE reell(str: ARRAY OF CHAR; VAR ok: BOOLEAN): RR;
PROCEDURE card (str: ARRAY OF CHAR): CARDINAL;

PROCEDURE RzuS(r: RR; n: CARDINAL; VAR str: ARRAY OF CHAR);
PROCEDURE CzuS(c: CARDINAL;         VAR str: ARRAY OF CHAR);

END ZKetten.
```

```
IMPLEMENTATION MODULE ZKetten;

IMPORT Strings, StrConv;

PROCEDURE gleich(VAR sl,s2: ARRAY OF CHAR): BOOLEAN;
VAR i: INDEX;
BEGIN
(* -------- eigene Version: lahm, läuft aber fast überall ------ *)
  i := 0;
  WHILE (i <= HIGH(sl)) AND (i <= HIGH(s2))
      AND (sl[i] = s2[i])
      AND (sl[i] <> 0C) AND (s2[i] <> 0C) DO
      INC(i)
  END;
  RETURN ((i > HIGH(sl)) OR (sl[i] = 0C)) AND ((i > HIGH(s2)) OR (s2[i] = 0C))
(* ------- Version Hänisch- SPC- TDI-Modula  -------->
  RETURN Strings.Compare(sl,s2) = 0  *)
(* ------- Version Megamax-Modula ------------>
  RETURN Strings.Compare(a,b) = Strings.equal  *)
END gleich;

PROCEDURE CopyCAP(VAR quelle, ziel: ARRAY OF CHAR);
(* Wandelt String 'quelle' in GROSSBUCHSTABEN nach 'ziel' um *)
VAR i: INDEX;
BEGIN
    i := 0;
```

```
    WHILE (i <= HIGH(quelle)) AND (i <= HIGH(ziel)) AND (quelle[i] <> 0C) DO
        ziel[i] := CAP(quelle[i]);
        INC(i) END;
    ziel[i] := 0C
END CopyCAP;

PROCEDURE reell(str: ARRAY OF CHAR; VAR ok: BOOLEAN): RR;
VAR pos: INDEX;
BEGIN
  pos := 0;
  RETURN StrConv.StrToReal(str, pos, ok);
END reell;

PROCEDURE card(str: ARRAY OF CHAR): CARDINAL;
VAR pos: INDEX; ok: BOOLEAN;
BEGIN
  pos := 0;
  RETURN StrConv.StrToCard(str, pos, ok)
END card;

PROCEDURE RzuS(r: RR; n: CARDINAL; VAR str: ARRAY OF CHAR);
VAR ss: Strings.String; ok: BOOLEAN;
BEGIN
  ss := StrConv.RealToStr(r, 15, n);
  Strings.Assign(ss, str, ok)
END RzuS;

PROCEDURE CzuS(c: CARDINAL; VAR str: ARRAY OF CHAR);
VAR ss: Strings.String; ok: BOOLEAN;
BEGIN
  ss := StrConv.CardToStr(c,5);
  Strings.Assign(ss, str, ok)
END CzuS;

END ZKetten.
```

5.2 Der Modul »Differenzierer«

Hier geht es um das algebraische Ableiten. Unsere Beispielfunktion `f(x)` mit dem Funktionsterm "`x^2 *sin x`" soll also die Zeichenkette für Ableitung `f'(x)=2 * X * SIN X + X^2 * COS X`" werden. Der Definitionsmodul enthält nur zwei Prozeduren:

```
DEFINITION MODULE Differenzierer;

FROM Parser IMPORT ParserPtr, BezString;

VAR
    DifferError: BOOLEAN;

PROCEDURE LerneAbleitung(StrFkt: BezString; StrAbleit: ARRAY OF CHAR);
PROCEDURE Ableitung(baum: ParserPtr): ParserPtr;

END Differenzierer.
```

Wie funktioniert nun das Ableiten im einzelnen?

Zunächst wird der Prozedur `Ableitung` unsere Funktion als Parserbaum übergeben. Mit diesem Baum ruft man die Prozedur `diff` auf, die die eigentliche Arbeit macht. Wenn Sie sich `diff` weiter unten im abgedruckten Implementationsmodul ansehen, erkennen Sie, daß hier nur folgende Ableitungsregeln implementiert sind:

```
1.   (-f)'        =-f'
2.   (f+g)'       =f'+g'
3.   (f-g)'       =f'-g'
4.   (f*g)'       =f'*g+f*g'
5.   (f/g)'       =(g*f'-g'*f)/(g*g)
6a.  (f^g)'       =(f^g)*(g'*LN(f)+g*f'/f)
6b.  (f^k)'       =k*f^(k-1) k konstant
7a.  x'           =1
7b.  k'           =0 k konstant
8.   (f(g(x)))'   =f'(g(x))*g'(x)
```

In den Fällen 1.–6a. wird die Bildung der Ableitung lediglich durch ein bis zwei Selbstaufrufe erledigt (Rekursion). Für den Fall 6a wird also die Logarithmusfunktion gebraucht. Für den Fall 8 muß für vordefinierte Funktionen die Ableitung bekannt sein. Zum Beispiel `sin' = cos`. Hierzu wird in der Prozedur `LerneAbleitung` die `ListeDerAbleitungen` durchsucht.

Im einzelnen wird aus unserem Beispielbaum:

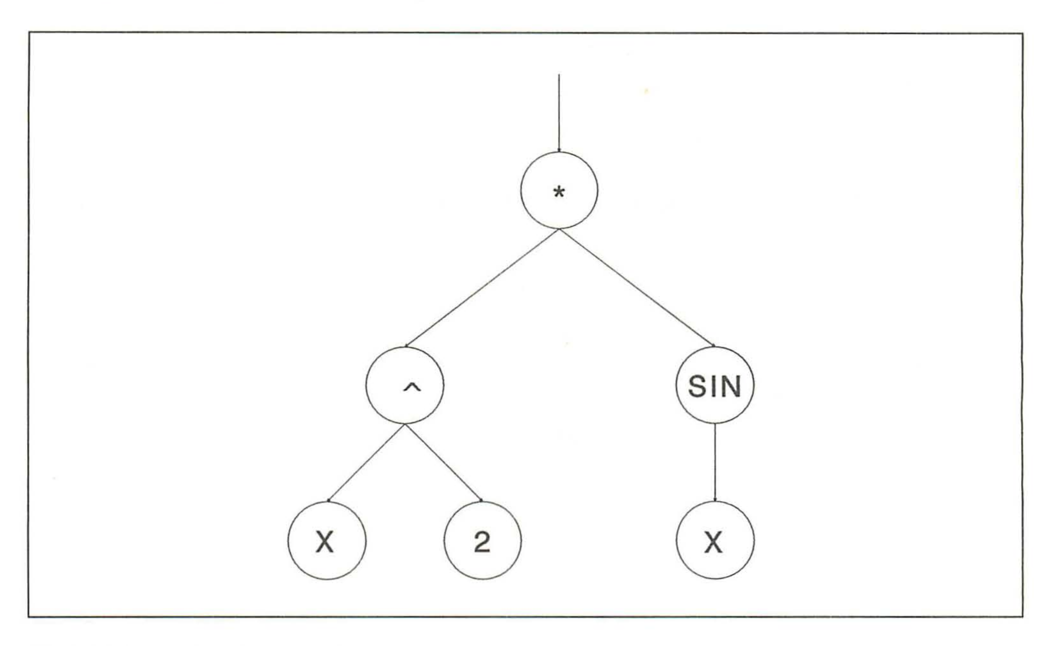


Bild 5.5: Funktionsbaum x sin x

Der Differenzierer macht daraus folgenden Baum:

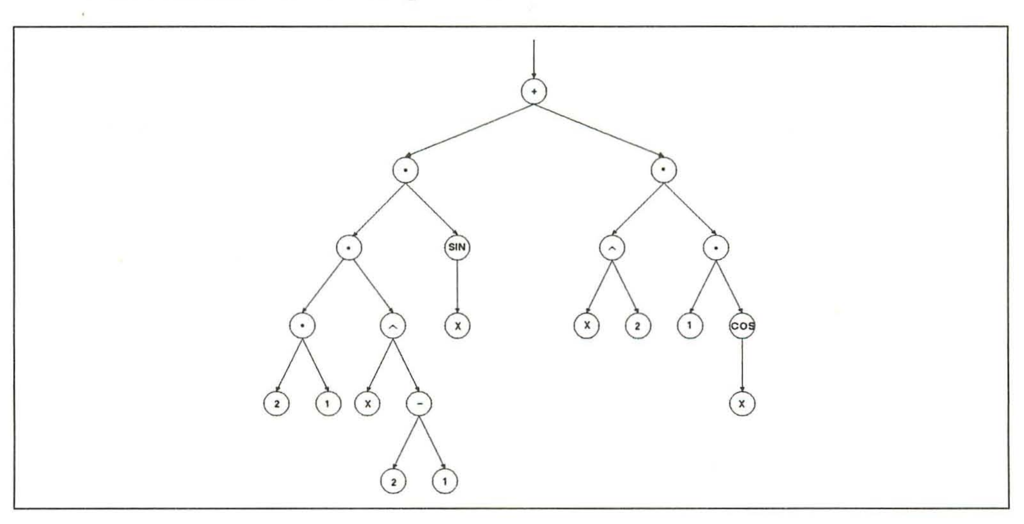


Bild 5.6: Funktionsbaum der nicht optimierten Ableitungsfunktion
f' (x) = (2 1)* (x ^(2-1))* sin x + x ^2* (1* cos x)*

Dieser Baum entspricht dem Term (2*1)*(x^(2-1))*sin x +x^2*(1*cos x) für die Ableitungsfunktion.

Dies ist sicherlich noch nicht die optimale Form

```
2 * X * SIN X + X ^ 2 * COS X,
```

weshalb noch ein »Optimierer« nachgeschaltet wird, um den Term zu kürzen. Doch zunächst zum Implementationsmodul `Differenzierer`:

```
IMPLEMENTATION MODULE Differenzierer;

FROM SYSTEM     IMPORT TSIZE;
FROM Parser     IMPORT
    SymType, RR, FunR1, BezString, BezPtr, ParserPtr,
    parse, NewNode, SyntaxError, HoleBezeichner, LoescheBaum;
FROM Optimierer IMPORT IsX, konstant, Optimiere, BaumCopy, ErsetzeCopy;
FROM Storage    IMPORT ALLOCATE;

TYPE
    DifferPtr = POINTER TO Differ;
    Differ = RECORD
                next: DifferPtr;
                Funktion: BezPtr;
                Ableitung: ParserPtr END;

VAR
    ListeDerAbleitungen: DifferPtr;
    LnFunktion: BezPtr;                           (* fürs Ableiten von a^b *)
```

```
PROCEDURE LerneAbleitung(StrFkt: BezString; StrAbleit: ARRAY OF CHAR);
VAR
    fkt: BezPtr;
    d: ParserPtr;
    r: DifferPtr;
BEGIN
    fkt := HoleBezeichner(StrFkt);
    IF fkt = NIL THEN HALT END;                     (* Funktion nicht bekannt *)
    IF fkt^.BezArt <> SymFunR1 THEN HALT END;               (* keine Funktion *)
    d := parse(StrAbleit);
    IF SyntaxError THEN HALT END;
    ALLOCATE(r, TSIZE(Differ));
    r^.next := ListeDerAbleitungen;                 (* in die Liste eintragen *)
    r^.Funktion := fkt;
    r^.Ableitung := d;
    ListeDerAbleitungen := r
END LerneAbleitung;

PROCEDURE HoleAbleitung(fkt: BezPtr): ParserPtr;
(* --- sucht die Ableitung einer Funktion aus der Liste --- *)
VAR r: DifferPtr;
BEGIN
    r := ListeDerAbleitungen;
    WHILE (r <> NIL) AND (r^.Funktion <> fkt) DO r := r^.next END;
    IF r = NIL THEN RETURN NIL ELSE RETURN r^.Ableitung END;
END HoleAbleitung;

PROCEDURE operator(op: SymType; a,b:ParserPtr): ParserPtr;
VAR p: ParserPtr;
```

```
BEGIN
    NewNode(p);
    p^.OperArt := op;
    p^.Operand1 := a;
    p^.Operand2 := b;
    RETURN p;
END operator;

PROCEDURE plus(a,b:ParserPtr): ParserPtr;
BEGIN
    RETURN operator(OpPlus, a,b)
END plus;

PROCEDURE minus(a,b:ParserPtr): ParserPtr;
BEGIN
    RETURN operator(OpMinus, a,b)
END minus;

PROCEDURE mal(a,b: ParserPtr): ParserPtr;
BEGIN
    RETURN operator(OpMal, a,b);
END mal;

PROCEDURE durch(a,b: ParserPtr): ParserPtr;
BEGIN
    RETURN operator(OpDurch, a,b)
END durch;

PROCEDURE hoch(a,b: ParserPtr): ParserPtr;
BEGIN
    RETURN operator(OpHoch, a,b)
END hoch;

PROCEDURE BaueLn(a: ParserPtr): ParserPtr;
VAR p: ParserPtr;
BEGIN
    IF LnFunktion = NIL THEN LnFunktion := HoleBezeichner("LN") END;
    IF LnFunktion = NIL THEN (* Eine Funktion "LN" muß da sein !!! *)
       DifferError := TRUE;
       RETURN NIL
    ELSE
        NewNode(p);
        p^.OperArt := SymFunR1;
        p^.BezFkt1 := LnFunktion;
```

```
        p^.Parameter := a;
        RETURN p
    END
END BaueLn;

PROCEDURE konst(x:RR): ParserPtr;
VAR p: ParserPtr;
BEGIN
    NewNode(p);
    p^.OperArt := SymKoRR;
    p^.KoRR := x;
    RETURN p
END konst;

PROCEDURE diff(p: ParserPtr): ParserPtr;
VAR d: ParserPtr;
BEGIN
    IF DifferError THEN RETURN NIL END;
    WITH p^
    DO CASE OperArt OF
            OpNeg:
                RETURN minus(konst(0.0), diff(Operand)) |
            OpMinus:
                RETURN minus(diff(Operand1), diff(Operand2)) |
            OpPlus:
                RETURN plus(diff(Operand1), diff(Operand2)) |
            OpMal:
                (* --- f(/g)' = (f'*g - f*g')/(g*g) --- *)
            RETURN plus(
                    mal(diff(Operand1), BaumCopy(Operand2)),
                    mal(BaumCopy(Operand1), diff(Operand2))) |
            OpDurch:
                (* --- f(*g)' = f'*g + f*g' --- *)
                RETURN durch(
                    minus (
                        mal(diff(Operand1),BaumCopy(Operand2)),
                        mal(BaumCopy(Operand1),diff(Operand2))),
                        mal (BaumCopy(Operand2), BaumCopy(Operand2))
                          ) |
            OpHoch:
                d := BaumCopy(Operand2);
                Optimiere(d);
                IF konstant(d) THEN
                (* ---- (f^k)' = k*f' * f^(k-1) ----- *)
```

```
                    RETURN mal(
                        mal(d, diff(Operandl)),
                        hoch(
                            BaumCopy(Operandl),
                            minus(BaumCopy(d), konst(1.0))
                            ) )
                ELSE
                (* ---- (f^g)' = f^g * (g'*ln f + g*f'/f)  ---- *)
                    RETURN mal(
                        hoch(BaumCopy(Operandl),BaumCopy(d)),
                        plus(
                            mal(diff(d),BaueLn(BaumCopy(Operandl))),
                            durch(mal(d, diff(Operandl)),
                                  BaumCopy(Operandl))
                            ) )
                END |
            SymKoRR, SymVarRR:
                IF IsX(p)
                    THEN RETURN konst(1.0)
                    ELSE RETURN konst(0.0) END |
            SymFunRl:
                d := HoleAbleitung(BezFktl);
                IF d = NIL THEN
                    DifferError := TRUE;
                    RETURN NIL
                ELSE
                    RETURN mal(diff(Parameter), ErsetzeCopy(d, Parameter))
    END
        ELSE
            DifferError := TRUE;
            RETURN NIL
        END
    END
END diff;

PROCEDURE Ableitung(baum: ParserPtr): ParserPtr;
VAR d: ParserPtr;
BEGIN
    IF baum = NIL THEN HALT END;
    DifferError := FALSE;
    d := diff(baum);
    IF DifferError THEN
        LoescheBaum(d)
```

```
    ELSE
        Optimiere(d)
    END;
    RETURN d
END Ableitung;

BEGIN
    ListeDerAbleitungen := NIL;
    LnFunktion := NIL
END Differenzierer.
```

5.3 Der Modul »Optimierer«

Wie erwähnt, soll der Optimierer den aus dem Differenzierer erhaltenen Baum nacharbeiten, so daß der Funktionsstring etwas gefälliger wird und schneller zu berechnen ist. Zum Beispiel soll aus dem Term »`1.0+2.0`« der Term »`3.0`« oder aus »`1.0*X`« kurz `X` gemacht werden. Hierzu sind verschiedene Regeln anzuwenden und auszuprobieren, ob es sich wirklich um eine Vereinfachung handelt.

Das System muß dazu erst einmal die Vereinfachungsregeln lernen. Zur Aufnahme dieser Regeln dient der im nächsten Abschnitt besprochene Modul `MatheLehrer`, der Möglichkeit bietet, dem System beliebige weitere Regeln beizubringen. Hier handelt es sich also um ein Programm der »Künstlichen Intelligenz«.

Doch zurück zum Optimierer. Die Regeln sind also schon vorgegeben, sie müssen also »gelernt« und angewandt werden. Der Definitionsmodul lautet:

```
DEFINITION MODULE Optimierer;

FROM Parser IMPORT ParserPtr;

TYPE
    UmformungPtr = POINTER TO Umformung;
    Umformung = RECORD
                    next: UmformungPtr;
                    alt,neu: ParserPtr
                END;

PROCEDURE LerneRegel(StrAlt, StrNeu: ARRAY OF CHAR);
PROCEDURE BaumGleich(b1,b2: ParserPtr): BOOLEAN;
PROCEDURE BaumCopy(alt: ParserPtr): ParserPtr;
```

```
PROCEDURE ErsetzeCopy(alt: ParserPtr; ex: ParserPtr): ParserPtr;
PROCEDURE IsX(node: ParserPtr): BOOLEAN;
PROCEDURE konstant(node: ParserPtr): BOOLEAN;
PROCEDURE Optimiere(VAR baum: ParserPtr);

END Optimierer.
```

Zur Erläuterung greifen wir wieder unser Beispiel auf. Es ging um die Ableitung der Funktion f(x) = $x^2 \cdot \sin x$. Der Differenzierer erstellte hieraus einen Baum, der dem Term `(2*1)*(x^(2-1))*sin x + x^2*(1*cos x)` entspricht. Der Optimierer baut hieraus nun folgenden Baum auf:

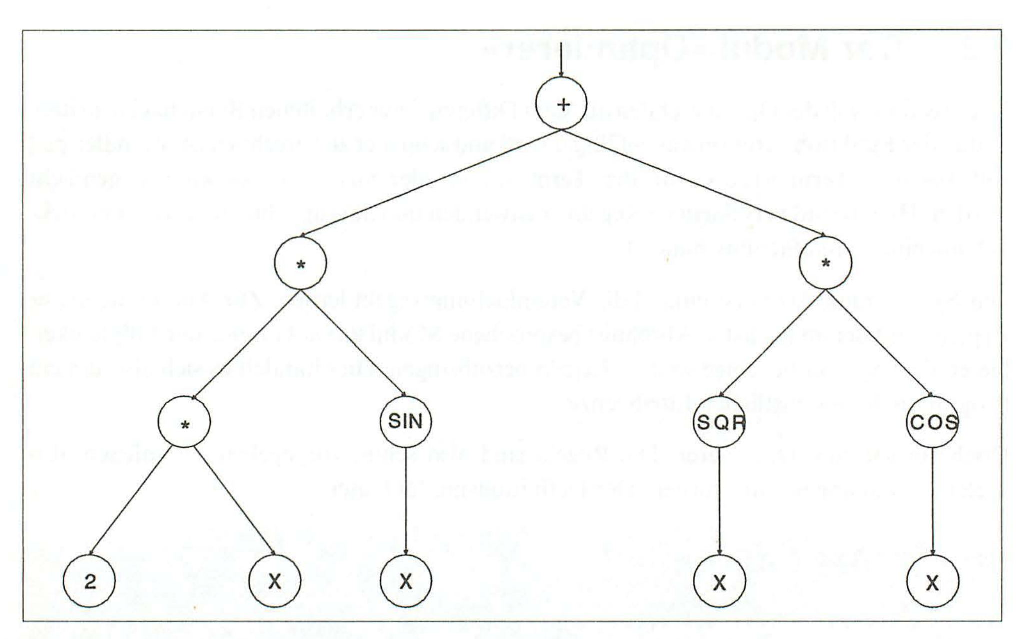


*Bild 5.7: Funktionsbaum 2x sin x + sqr(x)*cos x*

Dieser Baum entspricht nun dem String `2*x*sin x+sqr(x)*cos x`, wie man sich das wünscht.

Hier der Implementationsmodul:

```
IMPLEMENTATION MODULE Optimierer;

FROM SYSTEM IMPORT TSIZE;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
```

```
FROM Parser  IMPORT
    ParserPtr, SymType, NewNode, LoescheBaum, ArithmeticError,
    berechne, BezPtr, parse,
    HoleBezeichner, RR, SyntaxError, PrinTree;

VAR
    ListeDerRegeln: UmformungPtr;
    OptimierungWiederholen: BOOLEAN;
    X: BezPtr;

PROCEDURE konstant(node: ParserPtr): BOOLEAN;
BEGIN
    CASE node^.OperArt OF
        SymKoRR:
            RETURN TRUE
    ELSE
        RETURN FALSE
    END
END konstant;

PROCEDURE IsX(node: ParserPtr): BOOLEAN;
BEGIN
    RETURN (node^.OperArt = SymVarRR) AND (node^.Variable = X)
END IsX;

PROCEDURE SetzeKonstant(VAR node: ParserPtr);
VAR hw: RR;
BEGIN
    hw := berechne(node,999.999);
    LoescheBaum(node);
    NewNode(node);
    node^.OperArt := SymKoRR;
    node^.KoRR := hw;
    OptimierungWiederholen := TRUE
END SetzeKonstant;

PROCEDURE BaumCopy(alt: ParserPtr): ParserPtr;
VAR
    neu: ParserPtr;
BEGIN
    IF alt = NIL THEN HALT END;
    NewNode(neu);
    neu^.OperArt := alt^.OperArt;
    CASE alt^.OperArt OF
```

```
        OpNeg:
            neu^.Operand := BaumCopy(alt^.Operand) |
        OpMinus, OpPlus, OpDurch, OpMal, OpHoch:
            neu^.Operand1 := BaumCopy(alt^.Operand1);
            neu^.Operand2 := BaumCopy(alt^.Operand2) |
        SymKoRR, SymVarRR:
            neu^ := alt^ |
        SymFunR1:
            neu^.BezFkt1 := alt^.BezFkt1;
            neu^.Parameter := BaumCopy(alt^.Parameter)
    END;
    RETURN neu
END BaumCopy;

(* Kopie mit einsetzen von 'ex' in 'alt' ------------------ *)
PROCEDURE ErsetzeCopy(alt: ParserPtr; ex: ParserPtr): ParserPtr;
VAR neu: ParserPtr;
BEGIN
    IF alt = NIL THEN HALT END;
    IF IsX(alt) THEN RETURN BaumCopy(ex) END;
    NewNode(neu);
    neu^.OperArt := alt^.OperArt;
    CASE alt^.OperArt OF
        OpNeg:
            neu^.Operand := ErsetzeCopy(alt^.Operand, ex) |
        OpMinus, OpPlus, OpDurch, OpMal, OpHoch:
            neu^.Operand1 := ErsetzeCopy(alt^.Operand1, ex);
            neu^.Operand2 := ErsetzeCopy(alt^.Operand2, ex) |
        SymFunR1:
            neu^.BezFkt1 := alt^.BezFkt1;
            neu^.Parameter := ErsetzeCopy(alt^.Parameter, ex) |
        SymKoRR, SymVarRR:
            neu^ := alt^
    END;
    RETURN neu
END ErsetzeCopy;

PROCEDURE OpGleich(n1,n2: ParserPtr): BOOLEAN;
BEGIN
    IF n1^.OperArt = n2^.OperArt THEN
        CASE n1^.OperArt OF
            SymFunR1:
                RETURN n1^.BezFkt1 = n2^.BezFkt1 |
            SymKoRR:
```

```
                RETURN n1^.KoRR = n2^.KoRR |
            SymVarRR:
                RETURN n1^.Variable = n2^.Variable
        ELSE RETURN TRUE
        END
    ELSE
        IF konstant(n1) AND konstant(n2) THEN
            RETURN berechne(n1,123.456) = berechne(n2,321.654)
        ELSE
            RETURN FALSE
        END
    END
END OpGleich;

PROCEDURE BaumGleich(b1,b2: ParserPtr): BOOLEAN;
BEGIN
    IF b1 = NIL THEN HALT END;
    IF b2 = NIL THEN HALT END;
    IF NOT OpGleich(b1,b2) THEN RETURN FALSE END;
    CASE b1^.OperArt OF
        OpNeg:
            RETURN BaumGleich(b1^.Operand, b2^.Operand) |
        OpPlus, OpMal:                (* ---   kommutative Operatoren   --- *)
            RETURN (BaumGleich(b1^.Operand1, b2^.Operand1)
                AND BaumGleich(b1^.Operand2, b2^.Operand2))
               OR   (BaumGleich(b1^.Operand1, b2^.Operand2)
                AND BaumGleich(b1^.Operand2, b2^.Operand2)) |
        OpMinus, OpDurch, OpHoch:
            RETURN BaumGleich(b1^.Operand1, b2^.Operand1)
                AND BaumGleich(b1^.Operand2, b2^.Operand2) |
        SymFunR1:
            RETURN BaumGleich(b1^.Parameter, b2^.Parameter)
        ELSE
            RETURN TRUE
    END
END BaumGleich;

(* versucht, eine Regel aus der Liste anzuwenden ----------- *)
PROCEDURE NutzeRegel(VAR formel: ParserPtr; alt, neu: ParserPtr);
VAR FormX: ParserPtr;

    PROCEDURE TesteRegel(baum, vgl: ParserPtr): BOOLEAN;
    BEGIN
        IF IsX(vgl) THEN
```

```
            IF FormX = NIL
                THEN FormX := baum; RETURN TRUE
                ELSE RETURN BaumGleich(baum, FormX)
            END
        ELSE
            IF OpGleich(baum, vgl) THEN
                CASE baum^.OperArt OF
                    OpNeg:
                        RETURN TesteRegel(baum^.Operand,vgl^.Operand) |
                    OpPlus, OpMal:       (* ---- kommutative Operatoren ---- *)
                        RETURN (TesteRegel(baum^.Operand1, vgl^.Operand1)
                            AND TesteRegel(baum^.Operand2, vgl^.Operand2))
                         OR    (TesteRegel(baum^.Operand1, vgl^.Operand2)
                            AND TesteRegel(baum^.Operand2, vgl^.Operand1)) |
                    OpMinus, OpDurch, OpHoch:
                        RETURN TesteRegel(baum^.Operand1, vgl^.Operand1)
                            AND TesteRegel(baum^.Operand2, vgl^.Operand2) |
                    SymFunR1:
                        RETURN TesteRegel(baum^.Parameter, vgl^.Parameter)
                    ELSE
                        RETURN TRUE
                END
            ELSE
                RETURN FALSE
            END
        END
    END TesteRegel;

BEGIN (* NutzeRegel *)
    FormX := NIL;
    IF TesteRegel(formel, alt) THEN
        IF FormX <> NIL THEN FormX := BaumCopy(FormX) END;
        LoescheBaum(formel);                    (* ...hier geloescht werden *)
        formel := ErsetzeCopy(neu, FormX);
        IF FormX <> NIL THEN LoescheBaum(FormX) END;
        OptimierungWiederholen := TRUE
    END
END NutzeRegel;

(* alle Regeln durchprobieren ----------- *)
PROCEDURE ProbiereRegeln(VAR baum: ParserPtr);
VAR
    up: UmformungPtr;
    altbaum: ParserPtr;
```

```
BEGIN
    REPEAT
        altbaum := baum;
        up := ListeDerRegeln;
        WHILE up <> NIL DO
            NutzeRegel(baum, up^.alt, up^.neu);
            up := up^.next
        END
    UNTIL altbaum = baum          (* bis keine Regel mehr angewendet wurde *)
END ProbiereRegeln;

PROCEDURE optimize(VAR baum: ParserPtr);
BEGIN
    REPEAT
        IF ArithmeticError THEN RETURN END;
        (* -----  zuerst die UnterBaeume Optimieren   ----- *)
        WITH baum^ DO
            CASE baum^.OperArt OF
                OpNeg:
                    optimize(Operand) |
                OpMinus, OpPlus, OpDurch, OpMal, OpHoch:
                    optimize(Operand1);
                    optimize(Operand2) |
                SymFunR1:
                    optimize(Parameter)
            ELSE
                (* NIX, die anderen haben keine Unterbaeume *)
            END
        END;
        OptimierungWiederholen := FALSE;
        ProbiereRegeln(baum);
        (* ------  konstante Ausdruecke berechnen   ----- *)
        CASE baum^.OperArt OF
            OpNeg:
                IF konstant(baum^.Operand) THEN SetzeKonstant(baum) END |
            OpMinus, OpPlus, OpDurch, OpMal, OpHoch:
                IF konstant(baum^.Operand1) AND konstant(baum^.Operand2) THEN
                    SetzeKonstant(baum)
                END |
            SymFunR1:
                IF konstant(baum^.Parameter) THEN
                    SetzeKonstant(baum)
                END
        ELSE
```

```
            (* Nix *)
        END;
    UNTIL NOT OptimierungWiederholen;
END optimize;

PROCEDURE Optimiere(VAR f: ParserPtr);
BEGIN
    ArithmeticError := FALSE;
    optimize(f);
    IF ArithmeticError THEN LoescheBaum(f) END
END Optimiere;

PROCEDURE LerneRegel(StrAlt, StrNeu: ARRAY OF CHAR);
VAR regel: UmformungPtr;
BEGIN
    ALLOCATE(regel, TSIZE(Umformung));
    regel^.next := ListeDerRegeln;
    regel^.alt := parse(StrAlt); IF SyntaxError THEN HALT END;
    regel^.neu := parse(StrNeu); IF SyntaxError THEN HALT END;
    ListeDerRegeln := regel
END LerneRegel;

BEGIN
    X := HoleBezeichner("X");
    ListeDerRegeln := NIL;
END Optimierer.
```

5.4 Künstliche Intelligenz mit Modula: der Modul »MatheLehrer«

Der Definitionsmodul ist von überraschender Kürze:

```
DEFINITION MODULE MatheLehrer;
END MatheLehrer.
```

Es handelt sich also um eine reine Initialisierung. In `MatheLehrer` sind sämtliche Zuordnungen von Strings und den entsprechenden mathematischen Funktionen – wie "`SIN`" zu `sin` – in der Prozedur `LehreFunktionen` enthalten. Sie können diese Liste jederzeit mit eigenen Funktionen erweitern. Bei einer Erweiterung sollte auch der Prozedur `LehreAbleitungen` der Term der Ableitung hinzugefügt werden, sonst kann der Differenzierer diese Funktion nicht behandeln. Anschließend brauchen sie lediglich das Programm neu zu kompilieren, und schon können Sie die neue Funktion in einem Term benutzen.

`MatheLehrer` stellt nicht nur die Funktionszuordnungen für den Parser und die Ableitungszuordnungen für den Differenzierer bereit, sondern lehrt auch dem `Optimierer`, was er berücksichtigen soll. Hierzu wird eine Liste von Vereinfachungsregeln in der Prozedur `LehreRegeln` aufgelistet. Auch dieser Satz von Regeln ist beliebig erweiterbar. Der `MatheLehrer` lernt also von Ihnen (Professor) neue Regeln und kann sie dann an Parser, Optimierer und Differenzierer (Schüler) weitergeben. Bis jetzt sind folgende Variablen- und Funktionsbezeichner implementiert:

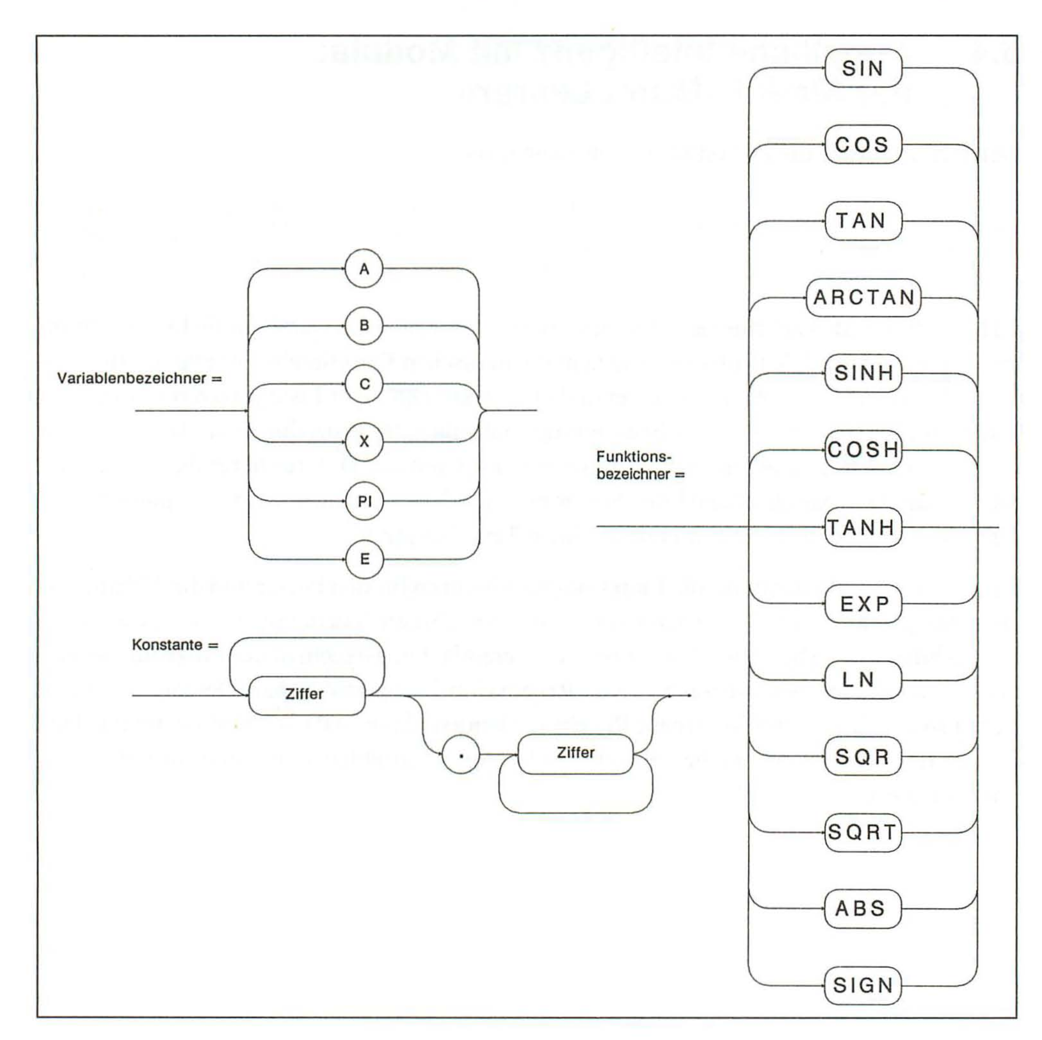


Bild 5.8: Syntaxdiagramme für Funktionsbezeichner, Variablenbezeichner und Konstante

Schauen Sie sich unbedingt den Implementationsmodul an, er ist ganz einfach aufgebaut!

```
IMPLEMENTATION MODULE MatheLehrer;

FROM MathLib0          IMPORT sin, cos, arctan, ln, exp, sqrt;
FROM Parser            IMPORT ArithmeticError, LerneFunktion,LerneVariable, RR;
FROM Optimierer        IMPORT LerneRegel;
FROM Differenzierer    IMPORT LerneAbleitung;
```

```
CONST schrott = 888.888;                    (* ein "undefiniertes" Ergebnis *)

PROCEDURE UnserTan(x: RR): RR;
VAR nenner: RR;
BEGIN
  nenner := cos(x);
  IF nenner = 0.0 THEN
    ArithmeticError := TRUE;
    RETURN schrott;
  ELSE
    RETURN sin(x) / nenner
  END
END UnserTan;

PROCEDURE Sh(x: RR): RR;
BEGIN
  RETURN (exp(x)-exp(-x)) / 2.0
END Sh;

PROCEDURE Ch(x: RR): RR;
BEGIN
  RETURN (exp(x)+exp(-x)) / 2.0
END Ch;

PROCEDURE Th(x: RR): RR;
BEGIN
  RETURN Sh(x) / Ch(x)
END Th;

PROCEDURE UnserLn(x: RR): RR;
BEGIN
  IF x <= 0.0 THEN
    ArithmeticError := TRUE;
    RETURN schrott;
  ELSE
    RETURN ln (x)
  END
END UnserLn;

PROCEDURE UnserSqrt(x: RR): RR;
BEGIN
  IF x < 0.0 THEN
    ArithmeticError := TRUE;
    RETURN schrott;
```

```
  ELSE
    RETURN sqrt(x)
  END
END UnserSqrt;

PROCEDURE Sqr(x: RR): RR;
BEGIN
  RETURN x*x
END Sqr;

PROCEDURE Abs(x: RR): RR;
BEGIN
  IF x < 0.0
    THEN RETURN -x
    ELSE RETURN x END
END Abs;

PROCEDURE Sign(x: RR): RR;
BEGIN
  IF x = 0.0 THEN RETURN 0.0
  ELSIF x > 0.0 THEN RETURN 1.0
  ELSE RETURN -1.0 END
END Sign;

PROCEDURE LehreKonstanten;
BEGIN
  LerneVariable("PI", 3.14159265358979328);
  LerneVariable("E", exp(1.0));
  LerneVariable("A", 0.0);
  LerneVariable("B", 0.0);
  LerneVariable("C", 0.0)
END LehreKonstanten;

PROCEDURE LehreFunktionen;
BEGIN
  LerneFunktion("SIN", sin);
  LerneFunktion("COS", cos);
  LerneFunktion("TAN", UnserTan);
  LerneFunktion("ARCTAN", arctan);
  LerneFunktion("SINH", Sh);
  LerneFunktion("COSH", Ch);
  LerneFunktion("TANH", Th);
  LerneFunktion("LN", UnserLn);
  LerneFunktion("EXP", exp);
```

```
  LerneFunktion("SQR", Sqr);
  LerneFunktion("SQRT", UnserSqrt);
  LerneFunktion("ABS", Abs);
  LerneFunktion("SIGN", Sign);
END LehreFunktionen;

PROCEDURE LehreRegeln;
BEGIN
  LerneRegel("TAN ARCTAN X", "X");
  LerneRegel("SIN X / COS X", "TAN X");
  LerneRegel("SQR SIN X + SQR COS X", "1");
  LerneRegel("SIN 0", "0");
  LerneRegel("COS 0", "1");
  LerneRegel("SIN PI", "0");
  LerneRegel("COS PI", "-1");
  LerneRegel("SQR COSH X - SQR SINH X", "1");
  LerneRegel("EXP LN X", "X");
  LerneRegel("LN EXP X", "X");
  LerneRegel("E^X", "EXP X");
  LerneRegel("SQRT SQR X", "ABS X");
  LerneRegel("SQR SQRT X", "X");
  LerneRegel("X*X", "SQR X");
  LerneRegel("X^2", "SQR X");
  LerneRegel("X^1", "X");
  LerneRegel("--X", "X");
  LerneRegel("0+X", "X");
  LerneRegel("X-0", "X");
  LerneRegel("0-X", "-X");
  LerneRegel("X-X", "0");
  LerneRegel("0*X", "0");
  LerneRegel("1*X", "X");
  LerneRegel("0/X", "0");
  LerneRegel("X/X", "1");
  LerneRegel("X/0", "1/0");
END LehreRegeln;

PROCEDURE LehreAbleitungen;
BEGIN
  LerneAbleitung("SIN", "COS X");
  LerneAbleitung("COS", "-SIN X");
  LerneAbleitung("TAN","1/SQRC COS X");
  LerneAbleitung("ARCTAN", "1/(1+SQR X)");
  LerneAbleitung("SINH", "COSH X");
  LerneAbleitung("COSH", "SINH X");
```

```
    LerneAbleitung("TANH", "1/ SQR(COSH X)");
    LerneAbleitung("EXP", "EXP X");
    LerneAbleitung("LN", "1/X");
    LerneAbleitung("SQRT", "1/(2*SQRT X)");
    LerneAbleitung("SQR", "2*X");
    LerneAbleitung("ABS", "SIGN X");
    LerneAbleitung("SIGN", "0");
  END LehreAbleitungen;

  BEGIN
    LehreFunktionen;
    LehreKonstanten;
    LehreRegeln;
    LehreAbleitungen;
  END MatheLehrer.
```

5.5 Optimiertes stabiles Integrationsverfahren

Vielleicht kennen Sie aus der Schulmathematik die Keplersche Faßregel

$$\text{(I)} \quad \int_a^b f(x)\,dx = \frac{h}{6}\left(f(a) + 4\,f\left(\frac{a+b}{2}\right) + f(b)\right) \quad \text{mit} \quad h = b - a$$

Diese Regel ermittelt das gesuchte Integral aus den Funktionswerten am Intervallanfang, -mitte und -ende. Wenn man das Verfahren nicht auf das gesamte Intervall anwendet, sondern dieses zuvor in 8 Teile zerlegt und auf die Teilintervalle anwendet, ist es relativ genau.

Nimmt man mehr Teilintervalle, ist das Ergebnis zwar exakter, die Rechenzeit wird aber höher. Das alte Lied! Wir greifen nun zu einem Trick: Wenn die Funktion nahezu konstant ist, reichen wenige Teilintervalle, um einen guten Wert für das Integral zu erhalten. Nur an den Stellen, wo sich der Funktionsgraph drastisch ändert, sind kleinere Teilintervalle angesagt. Man bringt also etwas Dynamik in die Sache! Die Vorteile der geringen Rechenzeit und des genauen Resultates werden kombiniert.

Aber wie soll man wissen, wo sich der Funktionsgraph stark ändert? Hierzu berechnen wir das Integral auf einem Teilintervall nach einer zweiten, noch besseren Formel. Diese Formel ist ähnlich aufgebaut wie die erste, verwendet aber pro Teilintervall drei Stützstellen:

$$\text{(II)} \quad \int_a^b f(x)\,dx = \frac{h}{90}\left(7\,f(a) + 32\,f\left(\frac{3a+b}{4}\right) + 12\,f\left(\frac{a+b}{2}\right) + 32\,f\left(\frac{a+3b}{4}\right) + 7\,f(b)\right)$$

Wenn der Funktionsverlauf nun relativ glatt ist, werden sich die errechneten Werte nach Formel `(I)` und nach Formel `(II)` kaum unterscheiden. Ansonsten halbieren wir das Intervall noch einmal, das ist alles!

Man startet mit acht Teilintervallen. Bliebe noch zu sagen, daß die Berechnungen so angelegt sind, das kein Funktionsaufruf doppelt erfolgt. Das Verfahren ist also sehr effizient.

```
DEFINITION MODULE Integrierer;

TYPE Funktion = PROCEDURE(REAL) : REAL;

PROCEDURE Integral( f : Funktion;
                    a,b,                              (* Integrationsgrenzen *)
                    Genauigkeit : REAL) : REAL;
END Integrierer.
```

```
IMPLEMENTATION MODULE Integrierer;

PROCEDURE Integral( f: Funktion; a,b,Genauigkeit : REAL) : REAL;
CONST
     minIntervalle = 8;
VAR
    x0,x1,xm,fa,fb,fm,I,h : REAL;
    j                     : CARDINAL;

  PROCEDURE TeilIntegral(a,m,b,fa,fm,fb : REAL);
  VAR I1,I2,h,m1,m2,fm1,fm2 : REAL;
  BEGIN
    h := b - a;
    m1 := (a + m)/2.0;
    fm1 := f(m1);
    m2 := (m + b)/2.0;
    fm2 := f(m2);
    I1 := (fa + 4.0*fm + fb) * h/6.0;                 (* Keplersche Faßregel *)
    I2 := (7.0*(fa + fb) + 32.0*(fm1 + fm2) + 12.0*fm) * h/90.0;
                                                       (* noch bessere Regel *)
    IF ABS(I1 - I2)>Genauigkeit THEN         (* wenn nicht in etwa gleich,... *)
```

```
        TeilIntegral(a,ml,m,fa,fml,fm);                 (* ... dann weiterteilen *)
        TeilIntegral(m,m2,b,fm,fm2,fb)
      ELSE I := I + I2 END                       (* den besseren Wert aufsummieren *)
    END TeilIntegral;

  BEGIN
    I := 0.0;
    h := (b - a)/FLOAT(minIntervalle);
    fb := f(a); xl := a;
    FOR j := 1 TO minIntervalle DO
      x0 := xl; fa := fb; xl := x0 + h;
      fb:=f(xl); xm := (x0+xl)/2.0; fm:=f(xm);
      TeilIntegral(x0,xm,xl,fa,fm,fb)
    END;
    RETURN I
  END Integral;

  END Integrierer.
```

5.6 Das komplette Programm »ModPlot«

Die wichtigsten Teile des Programmpaketes sind besprochen. Es fehlt nur noch die Benutzerschnittstelle:

- Die Menüleiste
- Diverse Dialogboxen zur Eingabe des Funktionsterms usw.
- Alarmboxen
- Die Grafikausgabe

Die Methoden für diese Programmteile wurden im 4. Kapitel besprochen, weshalb es hier keiner weiteren Erläuterungen bedarf. Lassen Sie das Programm einfach laufen, es liegt auf der Diskette als Stand-alone-Version vor und kann also vom Desktop aus angeklickt werden. Schauen Sie sich den Programmablauf in allen Einzelheiten an.

Wer noch Lust zum Forschen hat, für den sind die Benutzerschnittstellen-Module im folgenden aufgelistet. Zunächst der Hauptmodul, er enthält im wesentlichen nur Menüauswertung und verteilt die Arbeit auf andere Schultern, ein echter Manager...

```
MODULE ModPlot;

IMPORT AESForms, AESGraphics, GEMEnv;

FROM GEMGlobals   IMPORT PtrObjTree;
FROM AESMenus     IMPORT MenuBar, NormalTitle, CheckItem;
FROM AESEvents    IMPORT MessageBuffer, MessageEvent, menuSelected;
FROM AESResources IMPORT LoadResource, FreeResource, ResourceAddr,
                         ResourcePart;

IMPORT Grafik;
IMPORT Dialoge, GraphPlot;
IMPORT MatheLehrer;                         (* muss nur importiert werden! *)
IMPORT Plotrsc;          (* Die Resourcen-Definitionen für Menüs und Dialoge *)

PROCEDURE BehandleMenue;
VAR
    Menue : PtrObjTree;
    Nachricht: MessageBuffer;
BEGIN
  Menue := ResourceAddr(treeRsrc, Mntree);          (* Zeiger auf unser Menü *)
  MenuBar(Menue, TRUE);                                (* Das Menue anzeigen *)
  LOOP
    GraphPlot.Erneuern;
    MessageEvent(Nachricht);             (* warten auf Nachricht (Menü-Anwahl) *)
    CASE Nachricht.msgType OF
      menuSelected:                          (* Nachricht: Menüpunkt gewählt *)
          CASE Nachricht.selItem OF
             Mninfo   : Dialoge.Info |
             Mnfun    : Dialoge.FrageFunktion |
             Mnparms  : Dialoge.FrageParameter |
             Mnwerte  : GraphPlot.NeuerBereich |
             Mnhardcp: GraphPlot.Hardcopy |
             Mnneu    : GraphPlot.GraphLoeschen |
             Mnplot   : GraphPlot.PlotteFunktion |
             Mnintegr: Dialoge.FrageIntegral |
             Mndiffer: GraphPlot.PlotteAbleitung |
             Mnende   : EXIT
          END;
          NormalTitle(Menue, Nachricht.selTitle, TRUE) |
    ELSE
      (* andere Nachrichten berücksichtigen wir hier nicht *)
    END;
```

```
    END
END BehandleMenue;

VAR Knopf: CARDINAL;  (* Dummy *)

BEGIN
  Grafik.anmelden;                              (* Anmelden bei AES und VDI *)
  AESGraphics.GrafMouse(AESGraphics.bee, NIL);             (* Biene zeigen *)
  LoadResource("PLOTRSC.RSC");                      (* Resource- File Laden *)
  AESGraphics.GrafMouse(AESGraphics.arrow, NIL);            (* Pfeil zeigen *)
  IF GEMEnv.GemError() THEN                   (* Error: Alarmbox und abbruch *)
   AESForms.FormAlert(1,"[2][Das Resource-File|
                     PLOTRSC.RSC|konnte nicht geladen werden!][Pech]",
                     Knopf)
  ELSE                                     (* ---- nun kann es losgehen ---- *)
      GraphPlot.Init;
      BehandleMenue;                     (* Die Kommunikation mit dem Benutzer *)
      GraphPlot.Ende                          (* belegten Speicher freigeben *)
  END;
  Grafik.abmelden
END ModPlot.
```

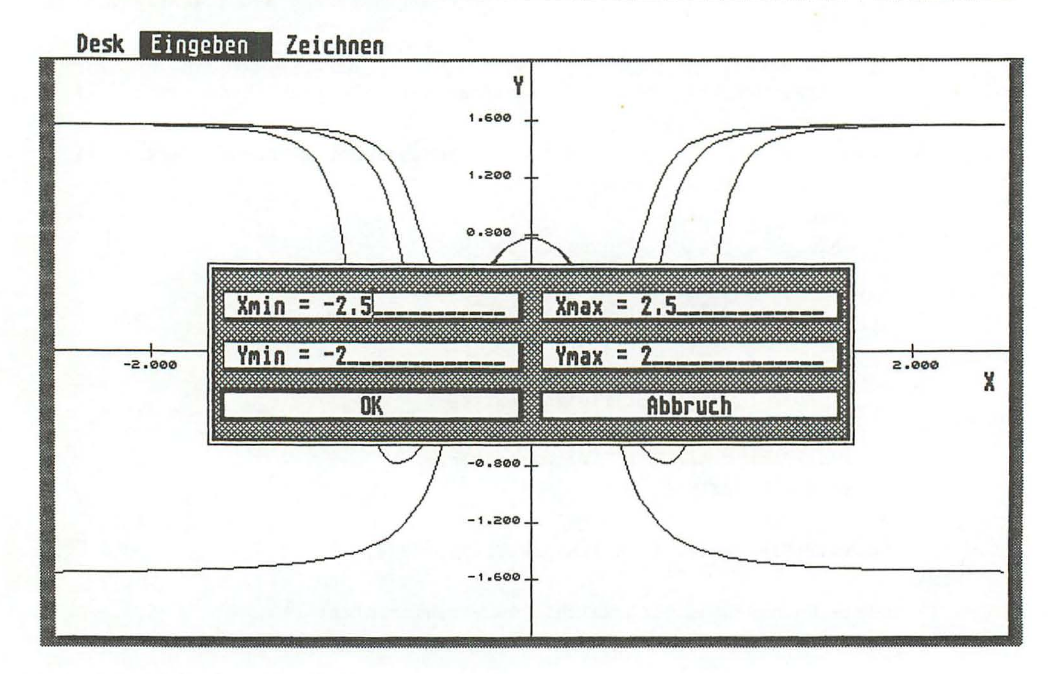


Bild 5.9: Dialogbox für die Parametereingabe

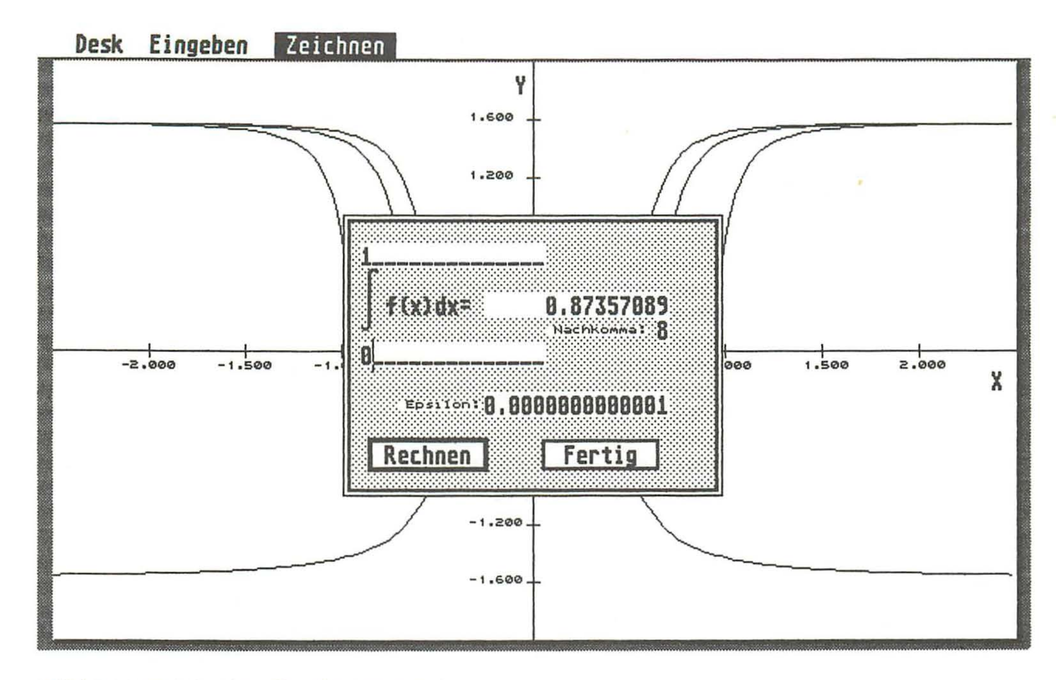


Bild 5.10: Dialogbox für das Integral

Der Eingabemodul steuert die diversen Dialogboxen. Weil der Anwender und das Programm sich einiges zu erzählen haben, gibt es davon sechs:

- Eingabe des Funktionsterms und der Parameter a,b und c
- Eingabe der Parameter separat
- Eingabe des Definitions- und Wertebereichs
- Ausgabe des Terms der Ableitungsfunktion
- Eingabe der Integrationsgrenzen und Integralausgabe
- Ausgabe der Information über ModPlot (Programmlogo)

Die Ausgabe des Strings der Ableitungsfunktion landet also in einer eigenen Box. Von hier aus hat man die Möglichkeit, diesen Funktionsterm als aktuellen in die Edierbox zu übernehmen, die Ableitung zu zeichnen oder die nächst höhere Ableitung zu ermitteln. Insgesamt erhält man vom Resource-Construktion-Set einen recht üppigen Datenmodul:

```
DEFINITION MODULE Plotrsc;

EXPORT   Mntree, Mninfo, Mnfun, Mnparms, Mnwerte, Mnhardcp,
         Mnende, Mnneu, Mnplot, Mndiffer, Mnintegr,
         Dfun, Dfunfkt, Dfuna, Dfunb, Dfunc, Dfunok, Dfunabbr,
```

```
          Dfunezgr, Dfunemsg, Dpar, Dpara, Dparb, Dparc,
          Dparok, Dparabbr, Dber, Dberxmin, Dberxmax, Dberymin,
          Dberymax, Dberok, Dberabbr, Dint, Dintvon, Dintbis,
          Dintrech, Dintende, Dinterg, Dintnako, Dinteps, Dinfo,
          Ddif, Ddifs1, Ddifs2, Ddifs3, Ddifs4, Ddifzahl,
          Ddifende, Ddifdif, Ddifplot;

CONST
  Mntree   =  0;  (* Menuebaum *)
  Mninfo   =  8;  (* STRING in Baum MNTREE *)
  Mnfun    = 17;  (* STRING in Baum MNTREE *)
  Mnparms  = 18;  (* STRING in Baum MNTREE *)
  Mnwerte  = 19;  (* STRING in Baum MNTREE *)
  Mnhardcp = 21;  (* STRING in Baum MNTREE *)
  Mnende   = 23;  (* STRING in Baum MNTREE *)
  Mnneu    = 25;  (* STRING in Baum MNTREE *)
  Mnplot   = 26;  (* STRING in Baum MNTREE *)
  Mndiffer = 28;  (* STRING in Baum MNTREE *)
  Mnintegr = 29;  (* STRING in Baum MNTREE *)
  Dfun     =  1;  (* Formular/Dialog *)
  Dfunfkt  =  4;  (* FBOXTEXT in Baum DFUN *)
  Dfuna    =  5;  (* FBOXTEXT in Baum DFUN *)
  Dfunb    =  6;  (* FBOXTEXT in Baum DFUN *)
  Dfunc    =  7;  (* FBOXTEXT in Baum DFUN *)
  Dfunok   =  8;  (* BUTTON in Baum DFUN *)
  Dfunabbr =  9;  (* BUTTON in Baum DFUN *)
  Dfunezgr = 10;  (* TEXT in Baum DFUN *)
  Dfunemsg = 11;  (* BOXTEXT in Baum DFUN *)
  Dpar     =  2;  (* Formular/Dialog *)
  Dpara    =  1;  (* FBOXTEXT in Baum DPAR *)
  Dparb    =  2;  (* FBOXTEXT in Baum DPAR *)
  Dparc    =  3;  (* FBOXTEXT in Baum DPAR *)
  Dparok   =  4;  (* BUTTON in Baum DPAR *)
  Dparabbr =  5;  (* BUTTON in Baum DPAR *)
  Dber     =  3;  (* Formular/Dialog *)
  Dberxmin =  1;  (* FBOXTEXT in Baum DBER *)
  Dberxmax =  2;  (* FBOXTEXT in Baum DBER *)
  Dberymin =  3;  (* FBOXTEXT in Baum DBER *)
  Dberymax =  4;  (* FBOXTEXT in Baum DBER *)
  Dberok   =  5;  (* BUTTON in Baum DBER *)
  Dberabbr =  6;  (* BUTTON in Baum DBER *)
  Dint     =  4;  (* Formular/Dialog *)
  Dintvon  =  3;  (* FTEXT in Baum DINT *)
  Dintbis  =  4;  (* FTEXT in Baum DINT *)
```

```
   Dintrech =   5;  (* BUTTON in Baum DINT *)
   Dintende =   6;  (* BUTTON in Baum DINT *)
   Dinterg  =   7;  (* TEXT in Baum DINT *)
   Dintnako =   9;  (* FTEXT in Baum DINT *)
   Dinteps  =  11;  (* FTEXT in Baum DINT *)
   Dinfo    =   5;  (* Formular/Dialog *)
   Ddif     =   6;  (* Formular/Dialog *)
   Ddifs1   =   2;  (* TEXT in Baum DDIF *)
   Ddifs2   =   3;  (* TEXT in Baum DDIF *)
   Ddifs3   =   4;  (* TEXT in Baum DDIF *)
   Ddifs4   =   5;  (* TEXT in Baum DDIF *)
   Ddifzahl =   7;  (* TEXT in Baum DDIF *)
   Ddifende =   8;  (* BUTTON in Baum DDIF *)
   Ddifdif  =   9;  (* BUTTON in Baum DDIF *)
   Ddifplot =  10;  (* BUTTON in Baum DDIF *)

END Plotrsc.
```

Wie gewohnt ist der Implementationsmodul leer, wie es sich für einen Datenmodul gehört.

```
IMPLEMENTATION MODULE Plotrsc;
(*$N+,M-*)
END Plotrsc.
```

Auf diesem Datenmodul baut nun die Verwaltung der verschiedenen Dialogboxen auf. Alles funktioniert nach dem im Abschnitt 4.8 besprochenen Rezepten. Die Länge kommt nur durch die Anzahl der Dialoge zustande.

```
DEFINITION MODULE Dialoge;

IMPORT GEMGlobals;
IMPORT Parser;

TYPE
    RR          = Parser.RR;
    ParserBaum  = Parser.ParserPtr;
    StrRR  = ARRAY [0..15] OF CHAR;              (* String für Reelle Zahlen *)
    StrFkt = ARRAY [0..60] OF CHAR;         (* String für einen Funktionsterm *)
    ParmStrings = RECORD a,b,c: StrRR END;
    BerStrings  = RECORD xMin, xMax, yMin, yMax: StrRR END;
```

```
VAR
    Parameter : RECORD
                   dlog: GEMGlobals.PtrObjTree;
                   StrS: ParmStrings;
                   BezP: RECORD a,b,c: Parser.BezPtr END
                 END;
    Funktion : RECORD
                   dlog: GEMGlobals.PtrObjTree;                (* Die Dialogbox *)
                   Baum: ParserBaum;                        (* Der Funktionsbaum *)
                   Stri: StrFkt;                           (* Der Funktionsstring *)
                   ErrZeiger: StrFkt;
                   ErrMeldung: ARRAY[0..30] OF CHAR;
                END;
    Bereich   : RECORD
                   dlog: GEMGlobals.PtrObjTree;
                   StrS: BerStrings;
                   xMin, xMax, yMin, yMax: RR
                END;
    Ableitung: RECORD
                   dlog: GEMGlobals.PtrObjTree;
                   Baum: ParserBaum;
                   Zahl: CARDINAL;
                 END;

  PROCEDURE Init;                                        (* Werte vorbesetzen *)
  PROCEDURE Ende;
  PROCEDURE Info;                                          (* Info- Box zeigen *)
  PROCEDURE FrageParameter;
  PROCEDURE FrageFunktion;
  PROCEDURE FrageBereich: BOOLEAN;
  PROCEDURE FrageIntegral;
  PROCEDURE FrageAbleitung: BOOLEAN;

  END Dialoge.
```

Hier der Implementationsmodul:

```
IMPLEMENTATION MODULE Dialoge;

FROM SYSTEM IMPORT ADR;
IMPORT GEMGlobals, GrafBase, ObjHandler;
IMPORT AESObjects, AESForms;
```

```
FROM AESResources IMPORT ResourceAddr, ResourcePart;
IMPORT Strings, StrConv, ZKetten;
IMPORT Parser, Integrierer, Differenzierer;
FROM Optimierer IMPORT BaumCopy, Optimiere;
IMPORT Plotrsc;

VAR
    DlgBox: RECORD                 (* Dialogboxen für Integral und die Info-Box *)
              Integral, Info: GEMGlobals.PtrObjTree;
            END;

MODULE DialogVerwalter;

FROM AESForms   IMPORT FormCenter, FormDial, FormDialMode, FormDo;
FROM AESObjects IMPORT DrawObject;
FROM GEMGlobals IMPORT PtrObjTree, OStateSet, Root, MaxDepth;
FROM GrafBase   IMPORT Rectangle, Rect;
FROM ObjHandler IMPORT SetObjState, SetCurrObjTree,
                       GetTextStrings, AssignTextStrings, SetPtrChoice;

EXPORT DialogInit, DialogFuehren, DialogEnde, TexLesen, TexSchreiben;

VAR BoxKlein, BoxGross: Rectangle;

PROCEDURE DialogInit(dlog: PtrObjTree);
BEGIN
  BoxKlein := Rect(300,200,50,30);       (* ein kleines Rechteck in der Mitte *)
  BoxGross := FormCenter(dlog);                       (* Größe der Dialogbox *)
  FormDial(reserveForm, BoxKlein, BoxGross);       (* Bildschirm reservieren *)
  FormDial(growForm,    BoxKlein, BoxGross);          (* ZOOM klein ---> groß *)
  SetCurrObjTree(dlog, FALSE);
END DialogInit;

PROCEDURE DialogEnde(dlog: PtrObjTree);
BEGIN
  FormDial(shrinkForm,  BoxKlein, BoxGross);        (* ZOOM groß - --> klein *)
  FormDial(freeForm,    BoxKlein, BoxGross)         (* Bildschirm freigeben *)
END DialogEnde;

PROCEDURE DialogFuehren(dlog: PtrObjTree; VAR EndeKnopf: CARDINAL);
BEGIN
  DrawObject(dlog, Root, MaxDepth, BoxGross);         (* Dialog-Box zeichnen *)
  FormDo    (dlog, Root, EndeKnopf);                     (* Benutzereingaben *)
```

```
  SetObjState(EndeKnopf, OStateSet{});            (* Ende-Knopf wieder normal *)
END DialogFuehren;

(* Einen String aus der Dialogbox lesen ---------------- *)
PROCEDURE TexLesen(Objekt: CARDINAL; VAR str: ARRAY OF CHAR);
VAR dummy : ARRAY[0..80] OF CHAR;
BEGIN
  GetTextStrings(Objekt, str, dummy, dummy);
END TexLesen;

(* Einen String in die Dialogbox schreiben ------------- *)
PROCEDURE TexSchreiben(Objekt: CARDINAL; str: ARRAY OF CHAR);
BEGIN
  AssignTextStrings(Objekt, setOnly, str, noChange,'',noChange,'');
END TexSchreiben;

END DialogVerwalter;

(* Eine reelle Zahl aus der Dialogbox lesen ------------ *)
PROCEDURE ReelleLesen(Objekt: CARDINAL; VAR ok: BOOLEAN): RR;
VAR str: StrRR;
    wert: RR;
BEGIN
  TexLesen(Objekt, str);
  wert := ZKetten.reell(str, ok);
  IF NOT ok THEN TexSchreiben(Objekt, "<???>") END;
  RETURN wert
END ReelleLesen;

PROCEDURE Info;                         (* <--- Anzeigen der Informations-Box *)
VAR Knopf: CARDINAL;
BEGIN
  DialogInit(DlgBox.Info);
  DialogFuehren(DlgBox.Info, Knopf);
  DialogEnde(DlgBox.Info)
END Info;

PROCEDURE LeseParameter: BOOLEAN;  (* ---- holt Parameter aus der Dialogbox *)
VAR ok1,ok2,ok3: BOOLEAN;
BEGIN
  WITH Parameter DO
    Parser.SetzeVariable(BezP.a, ZKetten.reell(StrS.a,ok1));
    Parser.SetzeVariable(BezP.b, ZKetten.reell(StrS.b,ok2));
    Parser.SetzeVariable(BezP.c, ZKetten.reell(StrS.c,ok3)) END;
```

```
  RETURN ok1 AND ok2 AND ok3
END LeseParameter;

PROCEDURE FrageParameter;       (* <---- neue Parameter über Dialogbox holen *)
VAR
    EndeKnopf: CARDINAL;
    AlteParms: ParmStrings;
BEGIN
  AlteParms := Parameter.StrS;
  DialogInit(Parameter.dlog);                (* Speicher reservieren und Effekte *)
  REPEAT
    DialogFuehren(Parameter.dlog, EndeKnopf);              (* Benutzereingaben *)
    IF EndeKnopf = Dparabbr THEN Parameter.StrS := AlteParms END;
  UNTIL LeseParameter() OR (EndeKnopf = Dparabbr);
  DialogEnde(Parameter.dlog);
END FrageParameter;

PROCEDURE Melde(s: ARRAY OF CHAR);
VAR i : CARDINAL;
BEGIN
  TexSchreiben(Dfunemsg, s);
END Melde;

PROCEDURE ZeigeFehler;                         (* Fehler und Position anzeigen *)
VAR i: CARDINAL;
BEGIN
  FOR i := 0 TO Parser.ScanErrPos DO Funktion.ErrZeiger[i] := "-"
END;
  Funktion.ErrZeiger[Parser.ScanErrPos] := "^";
  Funktion.ErrZeiger[Parser.ScanErrPos+1] := 0C;
  TexSchreiben(Dfunezgr, Funktion.ErrZeiger);
  CASE Parser.ErrorArt OF
    Parser.errCharacter  : Melde("unerlaubtes Zeichen") |
    Parser.errBezeichner : Melde("unbekannter Bezeichner") |
    Parser.errKlammerZu  : Melde('")" erwartet') |
    Parser.errAusdruck   : Melde("Ausdruck erwartet") |
    Parser.errOperator   : Melde("Operator erwartet")
    ELSE  Melde("Syntax-Fehler")
  END
END ZeigeFehler;

PROCEDURE FrageFunktion;        (* <-------------- Funktion eingeben lassen *)
VAR
    AlteFkt: Parser.ParserPtr;
```

```
    AlteStr: StrFkt;
    AlteParms: ParmStrings;
    EndeKnopf: CARDINAL;
BEGIN
  AlteFkt := Funktion.Baum;                  (* Alte Werte merken (für Abbruch) *)
  AlteStr := Funktion.Stri;
  AlteParms := Parameter.StrS;
  DialogInit(Funktion.dlog);
  Melde("<Funktion und Parameter angeben>");
  TexSchreiben(Dfunezgr, "");
  LOOP
    DialogFuehren(Funktion.dlog, EndeKnopf);
    IF EndeKnopf = Dfunabbr THEN                          (* Abbruch gedrückt? *)
        Funktion.Baum := AlteFkt;                (* alte Werte wiederherstellen *)
        Funktion.Stri := AlteStr;
        Parameter.StrS := AlteParms;
        EXIT       (* -------------------------------> Abbruch ==> Fertig *)
    ELSE
        IF LeseParameter() THEN                              (* Parameter ok? *)
          IF ZKetten.gleich(AlteStr,Funktion.Stri) THEN   (* neue Funktion? *)
              EXIT   (* --------------------------> Parameter ok ==> Fertig *)
          ELSE                            (* neue Funktion eingegeben ==> parsen *)
              Funktion.Baum := Parser.parse(Funktion.Stri);
              IF NOT Parser.SyntaxError THEN                  (* Funktion ok? *)
                Parser.LoescheBaum(AlteFkt);            (* Alte Funktion löschen *)
                Parser.LoescheBaum(Ableitung.Baum);     (* Ableitung ungültig *)
                EXIT       (* --------------> Parms & Funktion ok ==> Fertig *)
              ELSE ZeigeFehler             (* Fehler in der Dialogbox anzeigen *)
              END
          END
        ELSE
            Melde("Parameter A,B oder C ungültig!")  (* in der Box anzeigen *)
        END
    END
  END;
  DialogEnde(Funktion.dlog)                              (* Dialogbox abmelden *)
END FrageFunktion;

PROCEDURE LeseBereich: BOOLEAN;           (* Bereich für X/Y aus der Box lesen *)
VAR
    s: StrRR;
    ok1, ok2, ok3, ok4: BOOLEAN;
BEGIN
  WITH Bereich DO
```

```
    xMin := ReelleLesen(Dberxmin, ok1);
    xMax := ReelleLesen(Dberxmax, ok2);
    yMin := ReelleLesen(Dberymin, ok3);
    yMax := ReelleLesen(Dberymax, ok4);
    RETURN ok1 AND ok2 AND ok3 AND ok4 AND (xMin<xMax) AND (yMin<yMax)
  END
END LeseBereich;

PROCEDURE FrageBereich: BOOLEAN;              (* neuen Bereich eingeben lassen *)
VAR
    EndeKnopf: CARDINAL;
    SxMin, SxMax, SyMin, SyMax: StrRR;          (* Strings für reelle Zahlen *)
BEGIN
  DialogInit(Bereich.dlog);
  TexLesen(Dberxmin, SxMin);                 (* Alte Werte merken, da sie bei ... *)
  TexLesen(Dberxmax, SxMax);                 (* Abbruch restauriert werden müssen *)
  TexLesen(Dberymin, SyMin);
  TexLesen(Dberymax, SyMax);
  REPEAT
    DialogFuehren(Bereich.dlog, EndeKnopf);
    IF EndeKnopf = Dberabbr THEN              (* Abbruch => Werte Restaurieren *)
      TexSchreiben(Dberxmin, SxMin);
      TexSchreiben(Dberxmax, SxMax);
      TexSchreiben(Dberymin, SyMin);
      TexSchreiben(Dberymax, SyMax)
    END
  UNTIL LeseBereich() OR (EndeKnopf = Dberabbr);
  DialogEnde(Bereich.dlog);
  RETURN EndeKnopf = Dberok
END FrageBereich;

PROCEDURE f(x: RR):RR;
BEGIN
  RETURN Parser.berechne(Funktion.Baum, x)
END f;

PROCEDURE FrageIntegral;           (* <-------------- Integral-Dialog führen *)
VAR
    dlog          : GEMGlobals.PtrObjTree;
    EndeKnopf     : CARDINAL;
    epsilon       : RR;
    nako          : CARDINAL;       (* Zahl der gewünschten Nachkommastellen *)
    von,bis, igr  : RR;
    ok1,ok2,ok3   : BOOLEAN;
```

```
    str          : StrRR;
BEGIN
  DialogInit(DlgBox.Integral);
  TexSchreiben(Dinterg,"<!>");
  LOOP
    DialogFuehren(DlgBox.Integral, EndeKnopf);
    IF EndeKnopf = Dintende THEN EXIT END;
    von     := ReelleLesen(Dintvon, ok1);      (* untere Integrationsgrenze *)
    bis     := ReelleLesen(Dintbis, ok2);      (* obere Integrationsgrenze *)
    epsilon := ReelleLesen(Dinteps, ok3);                  (* Genauigkeit *)
    TexLesen(Dintnako, str);                          (* Nachkommastellen *)
    nako := ZKetten.card(str);
    IF ok1 AND ok2 AND ok3 THEN
        igr := Integrierer.Integral(f, von, bis, epsilon);
        ZKetten.RzuS(igr, nako, str);
        TexSchreiben(Dinterg,str)
    ELSE
        TexSchreiben(Dinterg,"...?")
    END
  END;
  DialogEnde(DlgBox.Integral)
END FrageIntegral;

PROCEDURE FunktionSchreiben;     (* Funktionsterm in die Dialogbox schreiben *)
VAR s: StrFkt; ok: BOOLEAN;
    ss: ARRAY[0..4*60] OF CHAR;
BEGIN
  Parser.BaumZuString(Ableitung.Baum, ss);
  Strings.Copy(ss, 0*60, 60, s, ok); TexSchreiben(Ddifs1, s);
  Strings.Copy(ss, 1*60, 60, s, ok); TexSchreiben(Ddifs2, s);
  Strings.Copy(ss, 2*60, 60, s, ok); TexSchreiben(Ddifs3, s);
  Strings.Copy(ss, 3*60, 60, s, ok); TexSchreiben(Ddifs4, s);
END FunktionSchreiben;

PROCEDURE FrageAbleitung: BOOLEAN;
VAR
    NeueAbleitung: ParserBaum;
    ZahlStr: ARRAY[0..5] OF CHAR;
    EndeKnopf: CARDINAL;
BEGIN
  DialogInit(Ableitung.dlog);
  IF Ableitung.Baum = NIL THEN          (* Wenn noch nicht abgeleitet wurde... *)
      Ableitung.Baum := BaumCopy(Funktion.Baum);       (* Funktion übernehmen *)
      Optimiere(Ableitung.Baum);
```

```
      Ableitung.Zahl := 0 END;
    LOOP
      FunktionSchreiben;          (* Funktionsterm in die Dialogbox bringen *)
      ZKetten.CzuS(Ableitung.Zahl, ZahlStr);       (* Nummer der Ableitung *)
      TexSchreiben(Ddifzahl, ZahlStr);
      DialogFuehren(Ableitung.dlog, EndeKnopf);
      CASE EndeKnopf OF
        Ddifende, Ddifplot: EXIT    (* fertig oder plotten ---------> raus  *)
        Ddifdif:                              (* einmal ableiten, bitte *)
            NeueAbleitung := Differenzierer.Ableitung(Ableitung.Baum);
            Parser.LoescheBaum(Ableitung.Baum);       (* Alten Baum löschen *)
            Ableitung.Baum := NeueAbleitung;
            INC(Ableitung.Zahl)          (* die wievielte haben wir den nun? *)
      END;
    END;
    DialogEnde(Ableitung.dlog);
    RETURN EndeKnopf = Ddifplot
  END FrageAbleitung;

  PROCEDURE Init;      (* <------- Alle Initialisierungen und Vorbesetzungen *)
  BEGIN
    Parameter.dlog  := ResourceAddr(treeRsrc, Dpar);
    Funktion.dlog   := ResourceAddr(treeRsrc, Dfun);
    Bereich.dlog    := ResourceAddr(treeRsrc, Dber);
    Ableitung.dlog  := ResourceAddr(treeRsrc, Ddif);
    DlgBox.Integral := ResourceAddr(treeRsrc, Dint);
    DlgBox.Info     := ResourceAddr(treeRsrc, Dinfo);
    ObjHandler.SetCurrObjTree(Parameter.dlog, FALSE);
    ObjHandler.LinkTextString(Dpara, ADR(Parameter.StrS.a));
    ObjHandler.LinkTextString(Dparb, ADR(Parameter.StrS.b));
    ObjHandler.LinkTextString(Dparc, ADR(Parameter.StrS.c));
    ObjHandler.SetCurrObjTree(Funktion.dlog, FALSE);
    ObjHandler.LinkTextString(Dfuna, ADR(Parameter.StrS.a));
    ObjHandler.LinkTextString(Dfunb, ADR(Parameter.StrS.b));
    ObjHandler.LinkTextString(Dfunc, ADR(Parameter.StrS.c));
    ObjHandler.LinkTextString(Dfunfkt, ADR(Funktion.Stri));
    Parameter.StrS.a := "0";
    Parameter.StrS.b := "0";
    Parameter.StrS.c := "0";
    Funktion.Stri := "X";
    ObjHandler.SetCurrObjTree(Bereich.dlog, FALSE);
    TexSchreiben(Dberxmin, "-5");
    TexSchreiben(Dberxmax, "5");
    TexSchreiben(Dberymin, "-3");
```

```
    TexSchreiben(Dberymax, "3");
    IF NOT LeseBereich() THEN HALT END;           (* initialer Bereich ungültig *)
    ObjHandler.SetCurrObjTree(DlgBox.Integral, FALSE);
    TexSchreiben(Dintvon, "0");                              (* Integralgrenzen *)
    TexSchreiben(Dintbis, "1");
    Parameter.BezP.a := Parser.HoleBezeichner("A");
    Parameter.BezP.b := Parser.HoleBezeichner("B");
    Parameter.BezP.c := Parser.HoleBezeichner("C");
    IF NOT LeseParameter() THEN HALT END;          (* initiale Parameter falsch *)
    Funktion.Baum := Parser.parse(Funktion.Stri);
    IF Parser.SyntaxError THEN HALT END;        (* initiale Funktion fehlerhaft *)
    Ableitung.Baum := NIL;                                   (* gibt noch keine *)
  END Init;

  PROCEDURE Ende;
  BEGIN
    Parser.LoescheBaum(Funktion.Baum);                    (* Speicher freigeben *)
    Parser.LoescheBaum(Ableitung.Baum);
  END Ende;

  END Dialoge.
```

Zu guter Letzt der Modul zur Grafikausgabe. Hier muß man sich etwas mit den Polstellen der Funktionen herumärgern. Im großen und ganzen kann die Arbeit aber auf unseren Modul `GrafikWelt` abgewältzt werden. Insbesondere `AchsenKreuz` leistet wieder gute Dienste. Etwas lästig ist auch noch das Neuzeichnen der Grafik nach dem Abgang einer Dialogbox vom Bildfläche. Hierzu wird der Bildschirminhalt zwischengepuffert und anschließend schnell zurückgeschrieben. Dies erledigt ein separater Modul `Bildschirm`, der unten ausgedruckt wird. Zunächst die Grafikroutinen:

```
DEFINITION MODULE GraphPlot;

IMPORT Parser;

TYPE
    RR = Parser.RR;
    ParserBaum = Parser.ParserPtr;

PROCEDURE Init;                          (* muß immer zuerst aufgerufen werden *)
PROCEDURE Ende;
PROCEDURE Sichern;                     (* Bildschirm sichern fürs Restaurieren *)
PROCEDURE Erneuern;                                 (* Bildschirm restaurieren *)
```

```
PROCEDURE Hardcopy;
PROCEDURE GraphLoeschen;      (* Bildschirm löschen und Achsenkreuz zeichnen *)
PROCEDURE NeuerBereich;
PROCEDURE PlotteFunktion;
PROCEDURE PlotteAbleitung;

END GraphPlot.
```

Der Implementationsmodul folgt sogleich:

```
IMPLEMENTATION MODULE GraphPlot;

IMPORT Grafik,GrafikWelt, Bildschirm, XBIOS;
IMPORT Parser;
FROM VDIInputs IMPORT ShowCursor, HideCursor;
FROM Dialoge   IMPORT Funktion, Parameter, Bereich, Ableitung;
IMPORT Dialoge;

CONST
  PixX1 = 10;    PixX2 = 630;
  PixY1 = 20;    PixY2 = 392;

PROCEDURE Sichern;
BEGIN
  HideCursor(Grafik.Geraet);                        (* Maus unsichtbar *)
  Bildschirm.ZeilenRetten(PixY1, PixY2);          (* Bildschirm sichern *)
  ShowCursor(Grafik.Geraet, FALSE)               (* Maus wieder sichtbar *)
END Sichern;

PROCEDURE Erneuern;                 (*Brutaler, aber wirkungsvoller Redraw*)
BEGIN
  HideCursor(Grafik.Geraet);
  Bildschirm.ZeilenErneuern(PixY1,PixY2);
  ShowCursor(Grafik.Geraet, FALSE)
END Erneuern;

PROCEDURE Hardcopy;
BEGIN
  HideCursor(Grafik.Geraet);
  Erneuern;
  XBIOS.ScreenDump;
  ShowCursor(Grafik.Geraet, FALSE);
END Hardcopy;
```

```
PROCEDURE GraphLoeschen;
BEGIN
  WITH Bereich DO
    Grafik.Hintergrund;
    GrafikWelt.SetzeSkalierung(xMin, yMin, xMax, yMax);
    GrafikWelt.AchsenKreuz((xMax-xMin)/10.0, (yMax-yMin)/10.0, 3,3, "X","Y");
    Sichern;
  END
END GraphLoeschen;

PROCEDURE NeuerBereich;
BEGIN
  IF Dialoge.FrageBereich() THEN GraphLoeschen END
END NeuerBereich;

PROCEDURE Move(x,y: RR);
BEGIN
  Grafik.Move(GrafikWelt.KonvertX(x), GrafikWelt.KonvertY(y));
END Move;

PROCEDURE Draw(x,y:RR);
BEGIN
  Grafik.Draw(GrafikWelt.KonvertX(x), GrafikWelt.KonvertY(y));
END Draw;

PROCEDURE Plotten(Fkt: ParserBaum);
CONST
    Toleranz = 5.0;
    Schritte = 200.0;
VAR
    x,y, DeltaX    : RR;
    MinTol, MaxTol : RR;
    innen          : BOOLEAN;
BEGIN
  HideCursor(Grafik.Geraet);    (* Maus immer vor dem Malen verstecken *)
  Erneuern;                              (* Bildschirm muß sauber sein *)
  DeltaX := (Bereich.xMax-Bereich.xMin) / Schritte;
  MaxTol := Bereich.yMax + Toleranz * (Bereich.yMax-Bereich.yMin);
  MinTol := Bereich.yMin - Toleranz * (Bereich.yMax-Bereich.yMin);
  x := Bereich.xMin;
  innen := FALSE;
  WHILE x <= Bereich.xMax DO
    y := Parser.berechne(Fkt, x);     (* einen Funktionswert berechnen *)
    IF Parser.ArithmeticError THEN innen := FALSE
```

```
    ELSIF y < MinTol              THEN innen := FALSE
    ELSIF y > MaxTol              THEN innen := FALSE
    ELSE
           IF innen THEN Draw(x,y) ELSE Move(x,y) END;
           innen := TRUE
    END;
    x := x + DeltaX;
  END;
  Sichern;
  ShowCursor(Grafik.Geraet, FALSE);              (* Maus wieder sichtbar *)
END Plotten;

PROCEDURE PlotteFunktion;
BEGIN
  Plotten(Funktion.Baum)
END PlotteFunktion;

PROCEDURE PlotteAbleitung;
BEGIN
  IF Dialoge.FrageAbleitung() THEN Plotten(Ableitung.Baum) END;
END PlotteAbleitung;

PROCEDURE Init;
BEGIN
  Dialoge.Init;
  Grafik.SetzeBereich(PixX1,PixY1, PixX2,PixY2);
  GraphLoeschen;
END Init;

PROCEDURE Ende; BEGIN Dialoge.Ende END Ende;

END GraphPlot.
```

Nun noch der oben angesprochene Modul zur Pufferung und Rekonstruktion des Bildschirms:

```
DEFINITION MODULE Bildschirm;

CONST
    BytesProZeile = 80;
(* ---------------------------------------------------------------- --
 * Mit diesen Routinen läßt sich der Bildschirm wieder restaurieren,
 * nachdem er (z.B. durch eine Dialogbox) zerstört worden ist, wenn
```

```
 * man sich nicht großartig mit "Window-Redraw's" abgeben will.
 * Hier wird einfach der Bildschirmspeicher kopiert; das ist zwar
 * brutal, aber recht effektiv zu benutzen.
 / ..................................................................
*)

PROCEDURE ZeilenRetten(yMin, yMax: CARDINAL);
(* Rettet die Bildschirmzeilen yMin bis yMax
 * in einen gesonderten Speicherbereich
 *)

PROCEDURE ZeilenErneuern(yMin, yMax: CARDINAL);
(*
 * restauriert die mit 'ZeilenRetten' gesichterten Zeilen
 *)

END Bildschirm.
```

```
IMPLEMENTATION MODULE Bildschirm;

FROM SYSTEM IMPORT BYTE, ADR, ADDRESS;
FROM LowLevel IMPORT CopyN;
IMPORT XBIOS;

TYPE
    Bild = ARRAY [0..31999] OF BYTE;   (* soll den Bildschirm aufnehmen *)

VAR
    BildschirmPtr : POINTER TO Bild;
    SchirmKopie: Bild;

PROCEDURE ZeilenRetten(yMin, yMax: CARDINAL);
VAR i: CARDINAL;
BEGIN
  CopyN(ADDRESS(BildschirmPtr) + LONG(yMin*BytesProZeile),
        ADR(SchirmKopie)       + LONG(yMin*BytesProZeile),
        (yMax-yMin+1) * BytesProZeile)
(* Entspricht in etwa: ------------------------------>
  FOR i := yMin*BytesProZeile TO yMax*BytesProZeile DO
     SchirmKopie[i] := BildschirmPtr^[i]
  END
<----------------------------------------------- *)
END ZeilenRetten;
```

```
PROCEDURE ZeilenErneuern(yMin, yMax: CARDINAL);
VAR i: CARDINAL;
BEGIN
    CopyN(ADR(SchirmKopie)          + LONG(yMin*BytesProZeile),
          ADDRESS(BildschirmPtr) + LONG(yMin*BytesProZeile),
          (yMax-yMin+1) * BytesProZeile)
(* Entspricht in etwa: ----------------------------->
 FOR i := yMin*BytesProZeile TO yMax*BytesProZeile DO
     BildschirmPtr^[i] := SchirmKopie[i]
  END
<------------------------------------------------------ *)
END ZeilenErneuern;

BEGIN
   BildschirmPtr := XBIOS.ScreenPhysicalBase();
END Bildschirm.
```

Damit sind alle Teile des Projekts `ModPlot` besprochen.

AUSBLICK

In diesem Buch haben wir uns das Ziel gesetzt, eine vollständige Einführung in Modula-2 zu geben, und Sie von der Schönheit und Stärke dieser Sprache – insbesondere der modularen Programmierung – zu überzeugen. Wir hoffen, daß Sie in den Bibliotheksmodulen Brauchbares für Ihre Programmierpraxis und in den zahlreichen Programmbeispielen Anregendes zur Weiterarbeit gefunden haben.

Zum Abschluß möchten wir noch einmal auf das Vorwort zurückkommen. Hier wurde berichtet, daß Modula-2 etwa seit zehn Jahren existiert. In der Zwischenzeit ist die Forschung an der ETH Zürich nicht stehengeblieben. Die mehrjährige Programmiererfahrung mit Modula haben *N. Wirth* und sein Team veranlaßt, Ende 1987 eine Weiterentwicklung namens »Oberon« zu veröffentlichen.

Oberon geht aus Modula mit einigen Erweiterungen und etlichen Streichungen hervor. Es handelt sich also insofern nicht um eine wesentlich neue Sprache, sondern sie bildet vielmehr das zunächst letzte Glied der Kette Algol, Pascal, Modula.

Neu an Oberon ist die Möglichkeit, Datentypen als Erweiterung bereits vorhandener zu definieren. Ein Beispiel:

```
TYPE T = RECORD x,y : INTEGER END;
```

Hiermit läßt sich als Typerweiterung definieren:

```
TYPE T1 = RECORD(T) z : REAL END;
```

Der Typ T1 enthält nun dieselben Komponenten wie T und zusätzlich noch z. T heißt Basistyp zu T1, T1 ist die Erweiterung zu T. Gemeinsame Komponenten des erweiterten Typs sind zuweisungskompatibel zu entsprechenden Variablen des Basistyps. Durch dieses Sprachkonstrukt ist es in Oberon möglich, nicht nur wie in Modula durch Prozeduren den Sprachkern zu erweitern, sondern auch durch Datentypen. Insbesondere für den Umgang mit Zeigervariablen eröffnet dies neue Möglichkeiten. Neben der Typerweiterung gibt es auch die Typinklusion. Oberon hat die fünf numerischen Datentypen

LONGINT, INTEGER, SHORINT	(ganzahlige Typen)
LONGREAL, REAL	(reelle Typen)

Hier ist folgende Inklusionsbeziehung gegeben:

```
LONGREAL ⊃ REAL ⊃ LONGINT ⊃ INTEGER ⊃ SHORTINT
```

$T \supset T'$ besagt dabei, daß eine Variable vom T' einer Variablen vom Typ T zugewiesen werden kann. Ist beispielsweise i vom Typ INTEGER, k vom Typ LONGINT und x vom Typ REAL, so sind die folgenden Zuweisungen zulässig:

```
k := i; x := k; k := k + i; x := x * 10 + i;
```

Nicht akzeptiert hingegen würde `i := k` oder `k := x`.

Gegenüber Modula wurden etliche Konstrukte weggelassen. Teilweise steht dies im Zusammenhang mit den genannten Erweiterungen, teilweise geschah es, um den Compiler zu entlasten und das System möglichst klein zu halten. Es gibt in Oberon keine varianten Verbunde und keine opaken Typen. Das Geheimnisprinzip kann jetzt dadurch gewahrt werden, daß im Definitionsteil nicht sämtliche Komponenten eines Verbundtypes deklariert werden müssen. Aufzählungstypen, Unterbereichstypen und der Datentyp `CARDINAL` fallen fort. Der kleinste Index eines Feldes ist stets Null. Zeiger können nur auf Verbunde oder Felder zeigen. Der Modul `SYSTEM` entfällt und mit ihm die Datentypen `ADDRESS` und `WORD`. Der Import von Bezeichner aus anderen Modulen darf nur noch qualifiziert geschehen. Es gibt keine lokalen Module mehr, keine `FOR`-Schleife, kein Coroutinenkonzept als Sprachbestandteil (möglicherweise aber als externen Modul) und die `WITH`-Anweisung hat eine geänderte Bedeutung.

Ansonsten bleibt aber alles beim Alten, so stimmen die Schlüsselwörter, die Standardfunktionen und die Syntaxregeln im großen und ganzen mit denen von Modula überein.

Sicherlich bieten nicht alle beschriebenen Streichungen Anlaß zu spontanem Jubel. Man wird sehen, ob die neuen Konzepte genügend Tragfähigkeit besitzen, derart viele Sprachkonstrukte adäquat zu ersetzen.

Bis die Sprache Oberon auch für unsere Rechnerkategorie kommt – wenn sie denn überhaupt kommt – wird sicher noch geraume Zeit vergehen. Eine Anfrage bei Professor Wirth Ende 1988 ergab, daß noch keine Implementierungen für Kleinrechner bekannt sind. Vielleicht dauert es wieder noch ein Jahrzehnt bis zu einer größeren Verbreitung. Bis dahin haben Sie mit Modula-2 eine leistungsstarke und elegante Sprache, die gemeinsam mit der guten Maschine Atari ST ein machtvolles Werkzeug für die Erstellung Ihrer Programmprojekte bietet!

Anhang

A. Literaturverzeichnis

[A]
R. AUMILLER, D. LUDA, G. MÖLLMANN:
»Atari ST GEM-Programmierung in C«,
Haar bei München 1987 (Markt & Technik)

[B]
G. BLASCHEK, G. POMBERGER, F. RITZINGER:
»Einführung in die Programmierung mit Modula-2«,
Berlin 1987 (Springer)

[C]
M. DAL CIN, J. LUTZ, T. RISSE:
»Programmierung in Modula-2«,
Stuttgart 1988 (3. Auflage, Teubner)

[D]
DITTRICH, ENGLISCH, SEVERIN:
»Das große Mega-ST-Buch«
Düsseldorf 88 (Data Becker)

[G]
J. GEISS, D. GEISS:
»Software-Entwicklung auf dem Atari ST«,
Heidelberg 1987 (Hüthing)

[H]
H. H. HAGER, J. SCHNUR:
»Prozeduren für Pascal-Programme«,
Paderborn 1988 (Schöningh)

[M]
F. MATHY:
»Programmierung von Grafik und Sound auf dem Atari ST«,
München 1987 (Markt & Technik)

[P]
H.-O. PEITGEN, P. H. RICHTER:
»The Beauty of Fractals«,
Berlin 1986 (Springer)

[V]
C. VIEILLEFOND:
»Programmierung des 68000«,
Düsseldorf 1985 (Sybex)

[W1]
N. WIRTH:
»Programmieren in Modula-2«,
Belin 1985 (Springer)

[W2]
N. WIRTH:
»Compilerbau«,
Stuttgart 1984 (Teubner)

[W3]
N. WIRTH:
»Systematisches Programmieren«,
Stuttgart 1985 (Teubner)

[W4]
N. WIRTH:
»From Modula to Oberon«,
Berichte der ETH Zürich 82, 1987

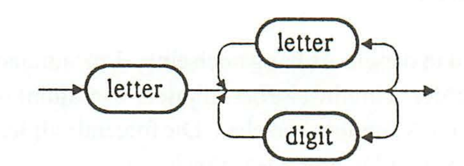
[W5]
N. WIRTH:
»The Programming Language Oberon«,
Berichte der ETH Zürich 82, 1987

[W6]
N. WIRTH:
»Algorithmen und Datenstrukturen mit Modula-2«,
Stuttgart, 1986 (Teubner)

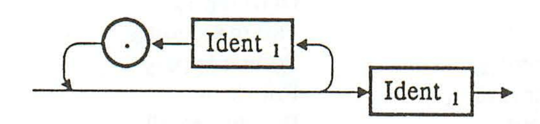
B. Syntaxdiagramme

Alle Syntax-Diagramme sind in diesem Anhang noch einmal zusammengefaßt; sie sollen bei der Klärung von Fragen nach der Schreibweise bestimmter Programmkonstruktionen helfen. Die Syntaxdiagramme sind mit Nummern versehen. Die folgende alphabetische Aufstellung soll das Auffinden eines gesuchten Diagrammes erleichtern.

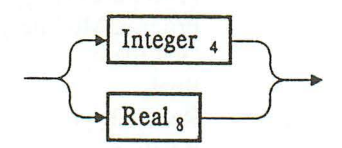
$Ident_1$



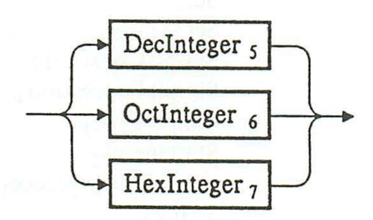
$QualIdent_2$



$Number_3$



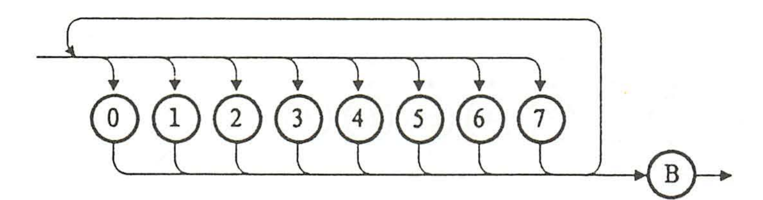
$Integer_4$



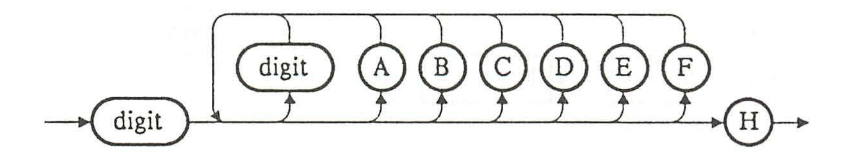
$DecInteger_5$



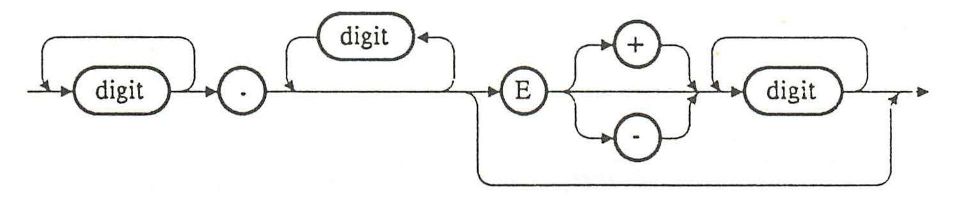
$OctInteger_6$



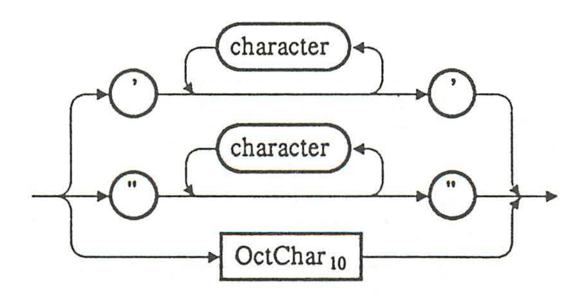
$HexInteger_7$



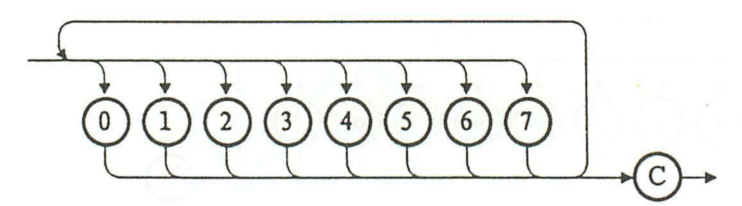
$Real_8$



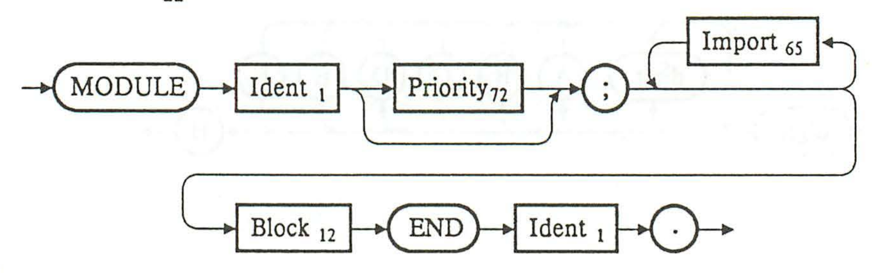
$String_9$



OctChar$_{10}$

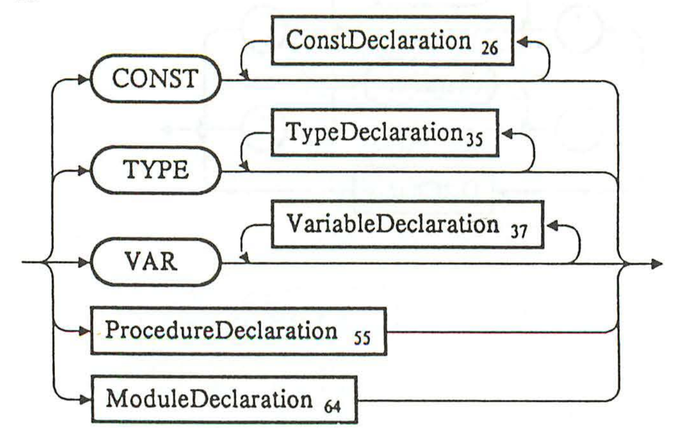


ProgramModule$_{11}$

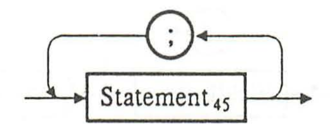


Block$_{12}$

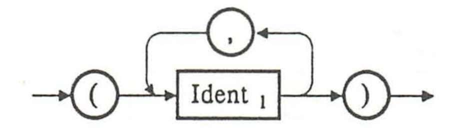
Declaration$_{13}$



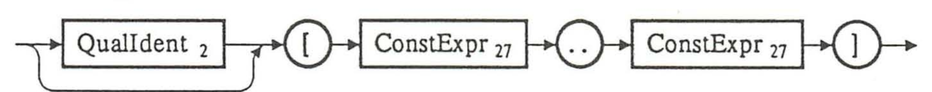
$StatementSequence_{14}$



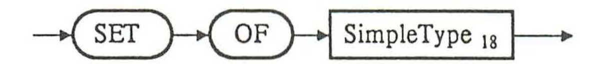
$Enumeration_{15}$



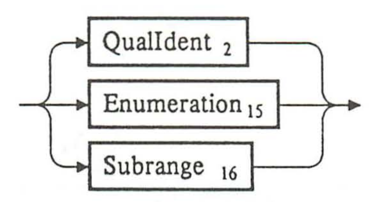
$Subrange_{16}$



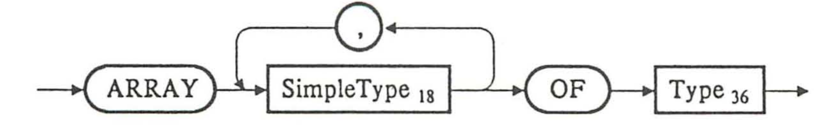
$SetType_{17}$



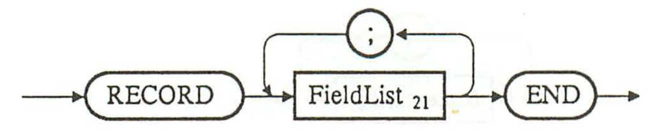
$SimpleType_{18}$



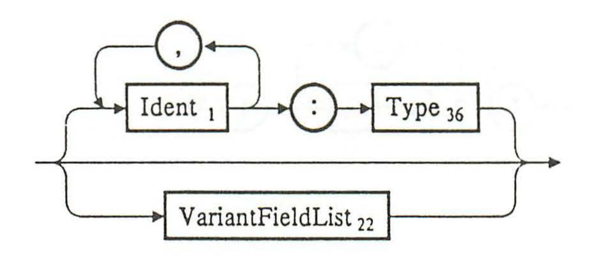
$ArrayType_{19}$



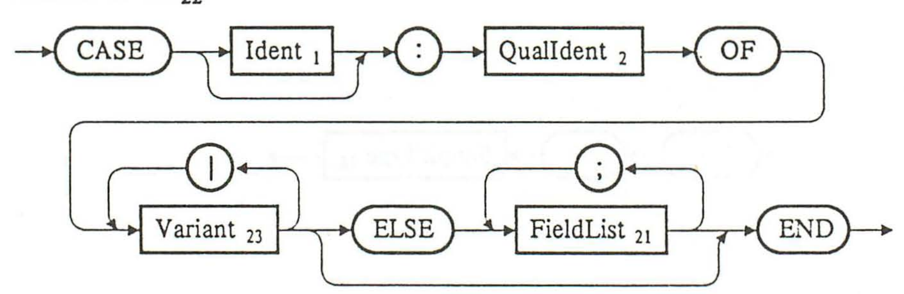
$RecordType_{20}$



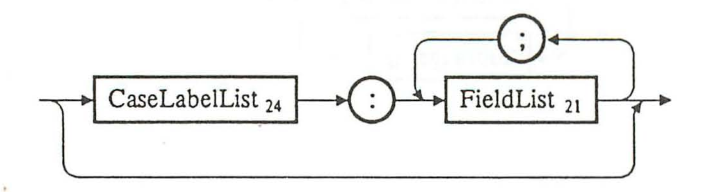
$FieldList_{21}$



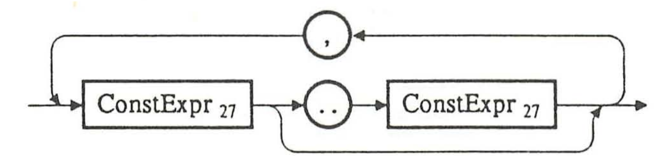
$VariantFieldList_{22}$



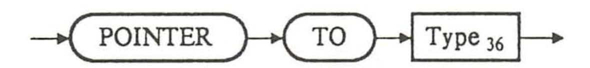
$Variant_{23}$



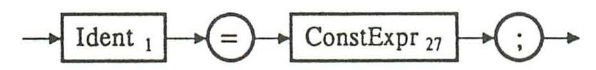
$CaseLabelList_{24}$



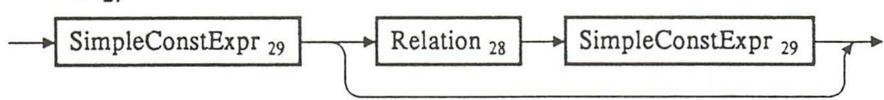
$PointerType_{25}$



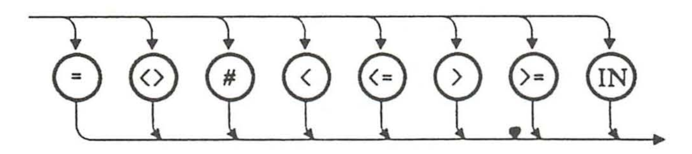
$ConstDeclaration_{26}$



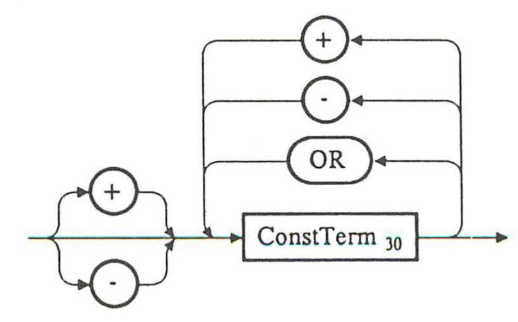
$ConstExpr_{27}$



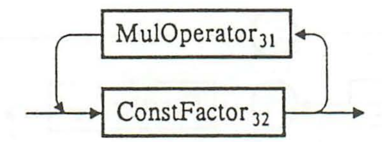
$Relation_{28}$



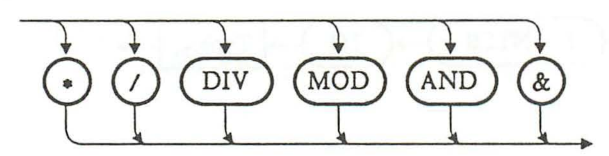
$SimpleConstExpr_{29}$



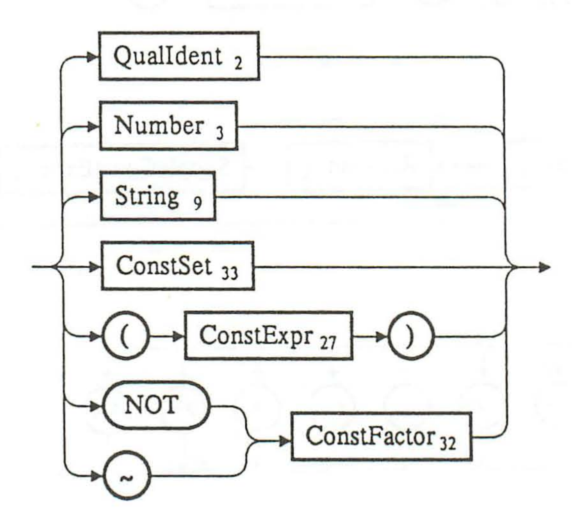
$ConstTerm_{30}$



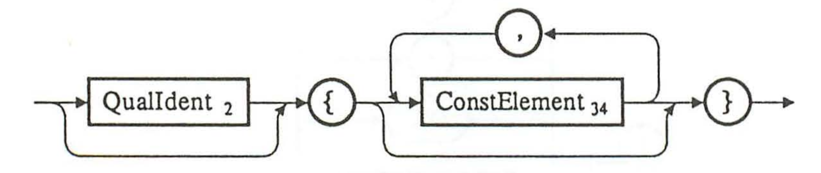
$MulOperator_{31}$



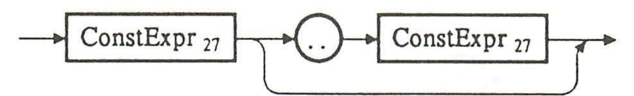
$ConstFactor_{32}$



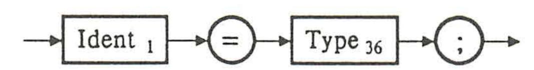
$ConstSet_{33}$



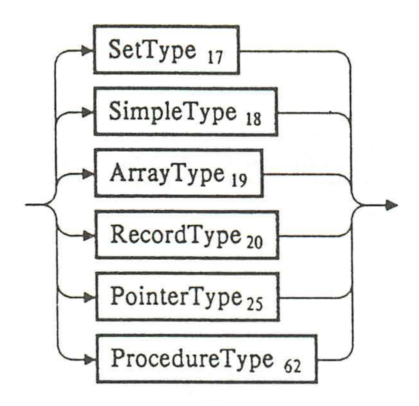
ConstElement$_{34}$



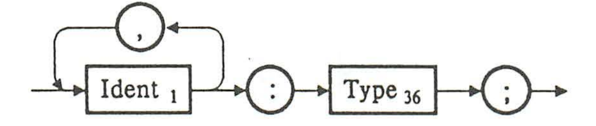
TypeDeclaration$_{35}$



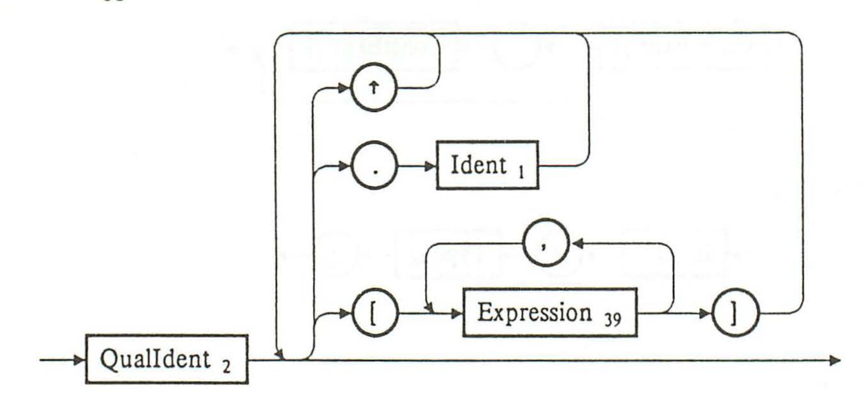
Type$_{36}$



VariableDeclaration$_{37}$



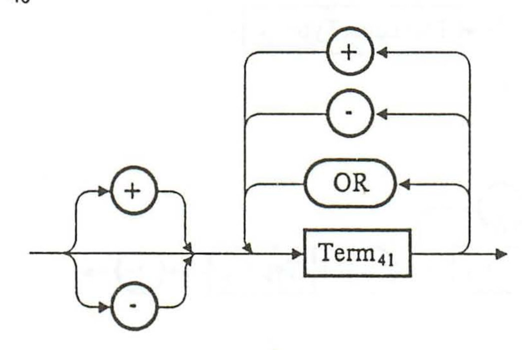
Designator$_{38}$

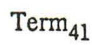


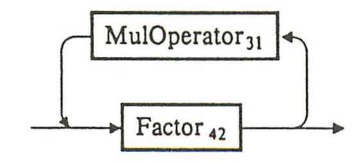
Expression$_{39}$

SimpleExpression $_{40}$ → Relation $_{28}$ → SimpleExpression $_{40}$

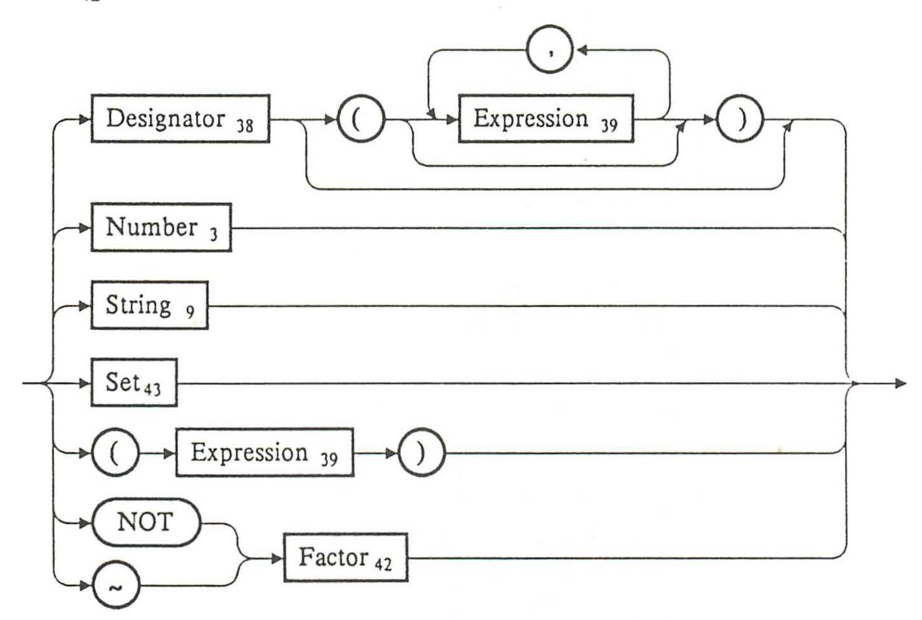
SimpleExpression$_{40}$



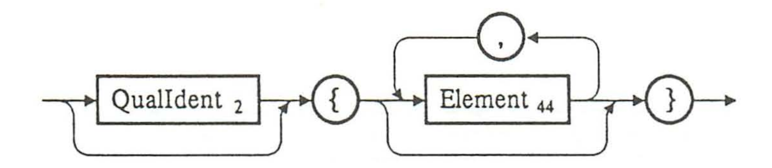




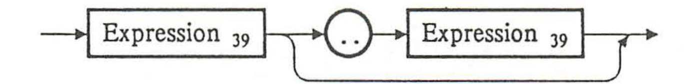
Factor$_{42}$



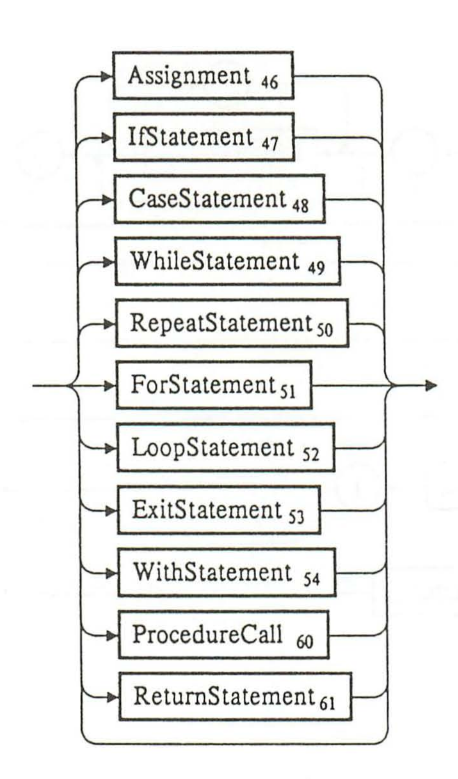
Set$_{43}$



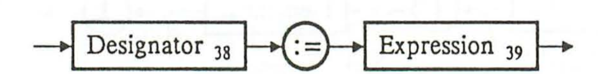
Element$_{44}$



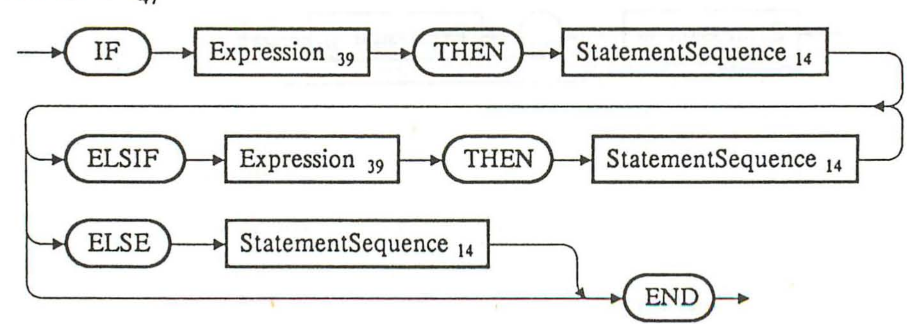
Statement$_{45}$



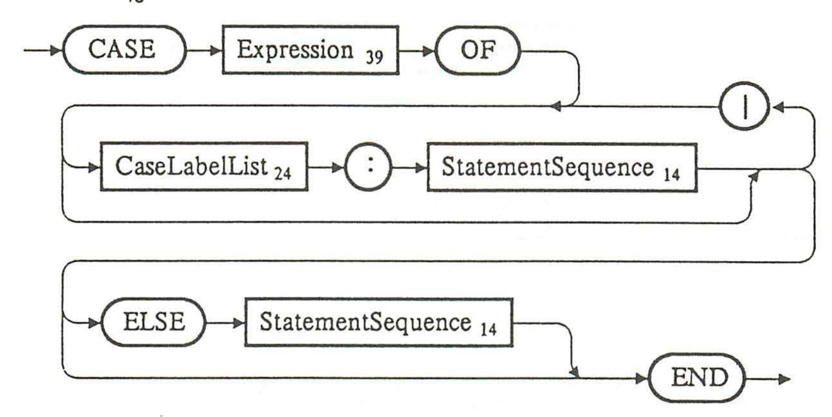
Assignment$_{46}$



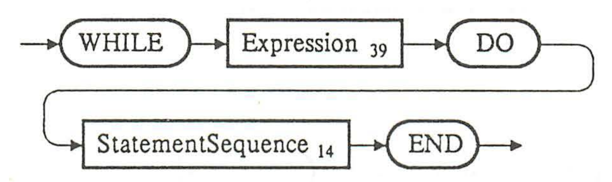
IfStatement$_{47}$



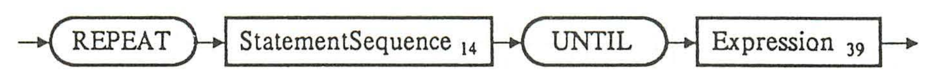
CaseStatement$_{48}$



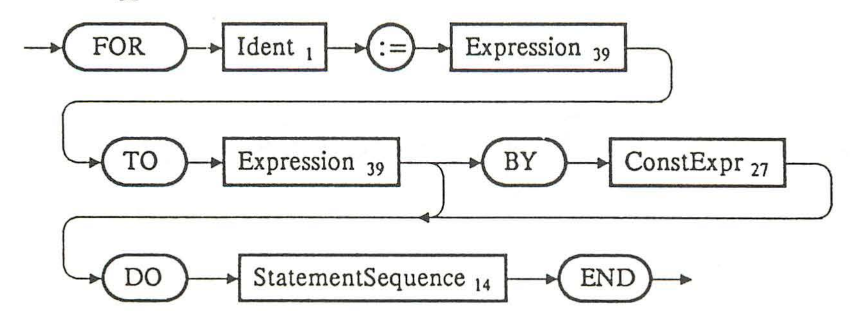
WhileStatement$_{49}$



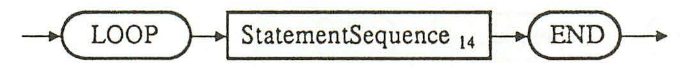
RepeatStatement$_{50}$



ForStatement$_{51}$



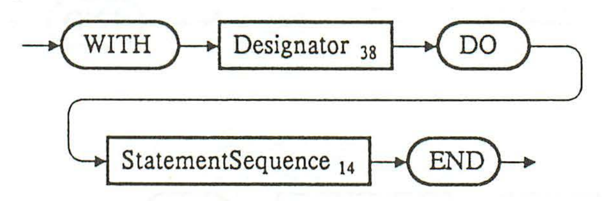
$LoopStatement_{52}$



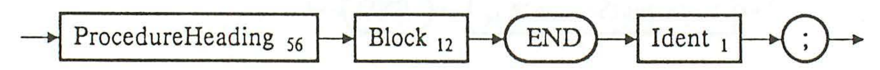
$ExitStatement_{53}$



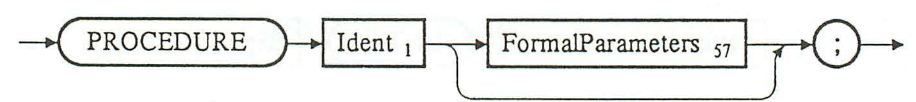
$WithStatement_{54}$



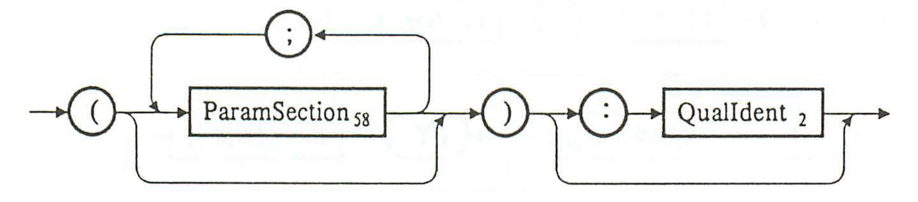
$ProcedureDeclaration_{55}$



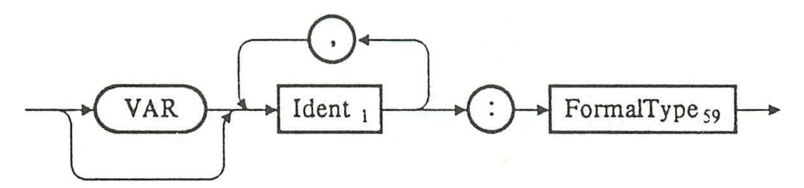
$ProcedureHeading_{56}$



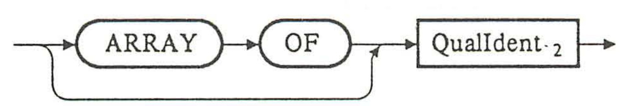
$FormalParameters_{57}$



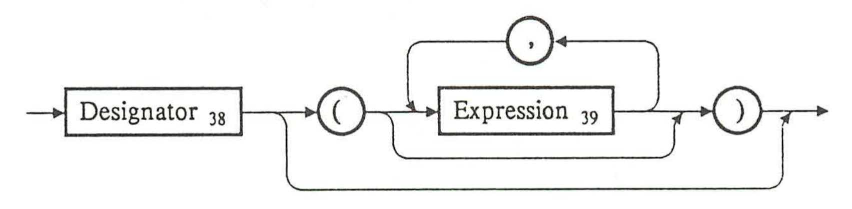
ParamSection$_{58}$



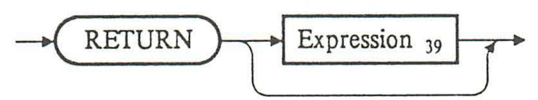
FormalType$_{59}$



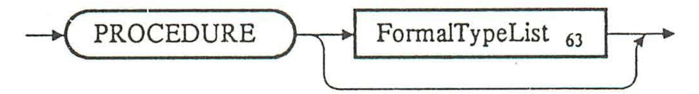
ProcedureCall$_{60}$



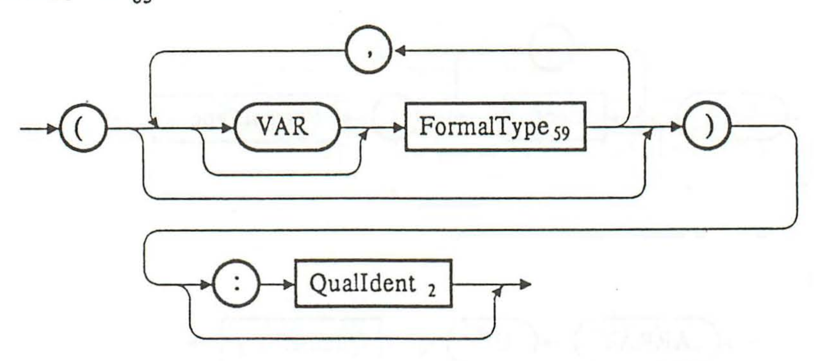
ReturnStatement$_{61}$



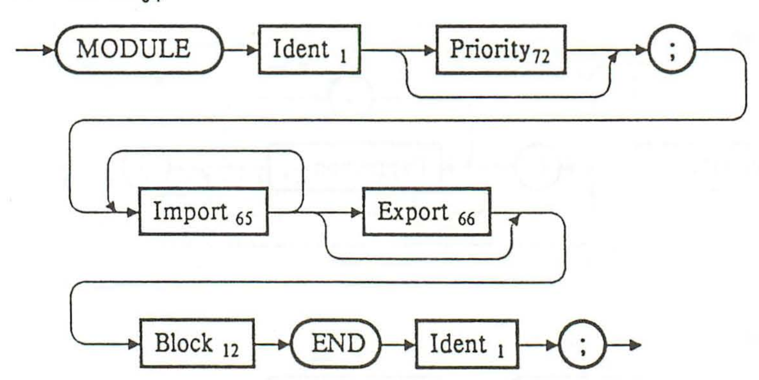
ProcedureType$_{62}$



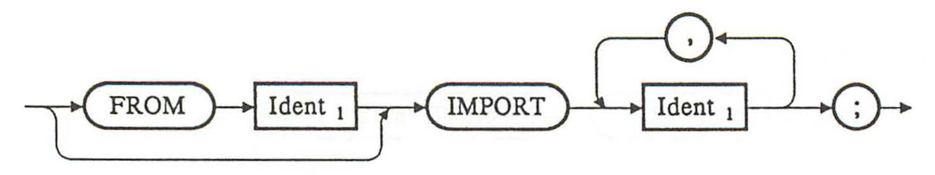
FormalTypeList$_{63}$



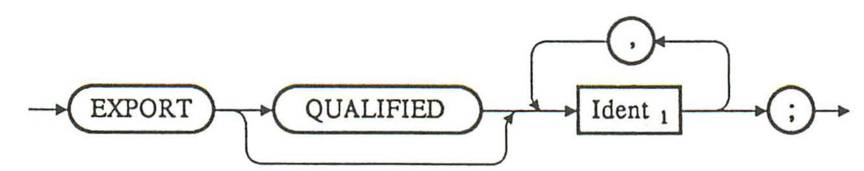
ModuleDeclaration$_{64}$



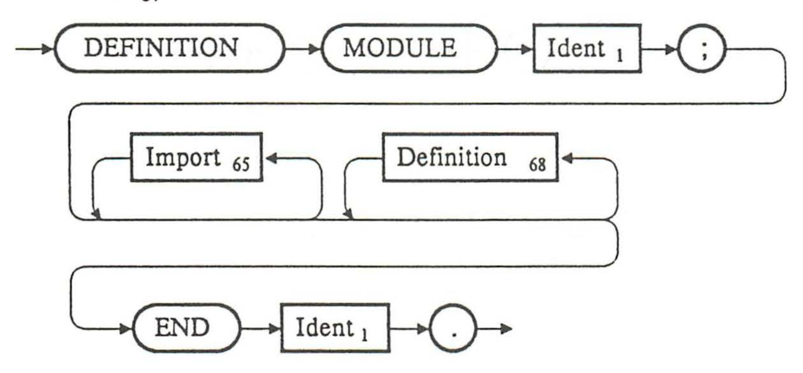
Import$_{65}$



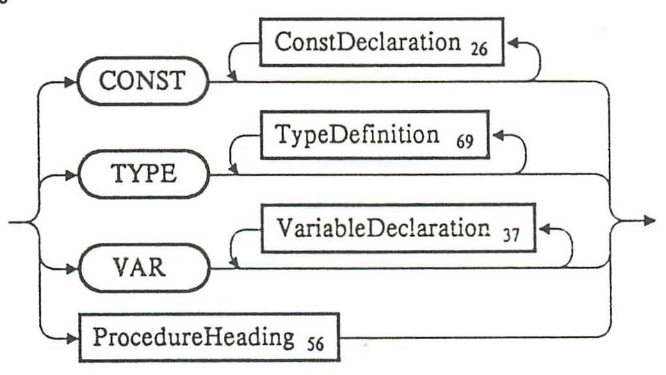
Export$_{66}$



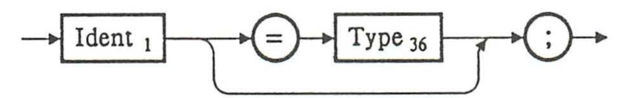
DefinitionModule$_{67}$



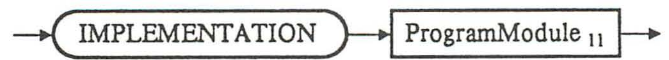
Definition$_{68}$



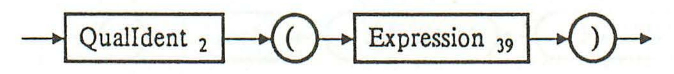
TypeDefinition$_{69}$



ImplementationModule$_{70}$



TypeTransfer$_{71}$



Priority$_{72}$



C. Liste der Befehle des Motorola-68000-Prozessors

Die Befehle sind in einem kompakten Format wie im folgenden Beispiel dargestellt:

ADD ea,Dn / ADD Dn,ea — Addiere
B W L — S+D —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	*	WL	*	*	*	*	*	*	*	*	*
D:			*	*	*	*	*	*	*			

In der ersten Zeile steht immer der Befehl in den erlaubten Syntax-Formen, gefolgt von einer kurzen Erklärung.

Dn ist ein beliebiges Datenregister.
An ist ein beliebiges Adreßregister.
Rn ist ein beliebiges Adreß- oder Datenregister.
S ist der Source-(Quell) Operand.
D ist der Destination-(Ziel) Operand.

#K ist eine Konstante
d ist die Adreßdistanz

In der zweiten Zeile stehen die erlaubten Operandengrößen (B, W, L). Darunter stehen die möglichen Adressierungsarten. Ein »*« heißt, daß alle vorgenannten Operandengrößen auch bei dieser Adressierung erlaubt sind. Ein oder zwei Buchstaben beschränken die Adressierungsart auf die durch die Buchstaben angedeuteten Operandengrößen.

cc in zum Beispiel DBcc heißt »Condition Code«. Seine Bedeutung ist auf der letzten Seite dieses Anhangs aufgeführt.

ABCD Dn,Dn / ABCD -(An),-(An) — Addiere BCD
B — S+D+X —> D

ADD ea,Dn / ADD Dn,ea — Addiere
B W L — S+D —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	WL	*	*	*	*	*	*	*	*	*	*
D:	*		*	*	*	*	*	*	*			

ADDA ea,An — Addiere Adresse

W L — S+D —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	*	*	*	*	*	*	*	*	*	*	*

Wortoperand wird wie bei EXT.L erweitert

ADDI #K,ea — Addiere Konstante

B W L — #K+D —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

ADDQ #K,ea — Addiere Konstante Quick (#K <=8)

B W L — #K+D —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*	WL	*	*	*	*	*	*	*			

ADDX Dn,Dn / ADDX -(An),-(An) — Addiere mit X-Flag

B W L — S+D+X —> D

AND ea,Dn / AND Dn,ea — Logisch UND

B W L — S AND D —> D

	DN	AN	(AN)	(AN)+	-(AN)	D(AN)	D(AN,RN)	$.W	$.L	d(PC)	d(PC,RN)	#
S:	*		*	*	*	*	*	*	*	*	*	*
D:			*	*	*	*	*	*	*			

ANDI #K,ea — Logisch UND mit Konstante

B W L — #K AND D —> D

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:*		*	*	*	*	*	*	*			

ANDI #K,CCR — Unde zu CCR
B — #K AND CCR —> CCR

ANDI #K,SR — Unde zu SR ! Privilegiert !
W — #K AND SR —> SR

ASL Dn,Dn / ASL #K,Dn / ASL ea — Arithmetisch links schieben
B W L — D n Bits geschoben —> D

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:		W	W	W	W	W	W	W			

0 wird nachgeschoben, herausgeschobenes Bit geht in das C- und X-Flag

---

ASR Dn,Dn / ASR #K,Dn / ASR ea — Arithmetisch rechts schieben
B W L — D n Bits geschoben —> D

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:		W	W	W	W	W	W	W			

MS-Bit schiebt, bleibt aber erhalten. Herausgeschobenes Bit geht in das C- und X-Flag

---

Bcc Label — Verzweige wenn cc (PC-relativ)
.B W — siehe cc-Tabelle
.S — PC+d —> PC

BCHG Dn,ea / BCHG #K,ea — Bit n Testen und ändern
B L — Bit-Test —> Z-Flag
Bit ändern

Wenn Source Dn: n=0..31, sonst 0..7. Wenn Destination im RAM, wird immer 1 Byte gelesen und n=n mod 8.

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:L		B	B	B	B	B	B	B			

BCLR Dn,ea / BCLR #K,ea — Bit n Testen und Löschen
B L — Bit-Test —> Z-Flag
Bit = 0

Wenn Source Dn: n=0..31, sonst 0..7. Wenn Destination im RAM, wird immer 1 Byte gelesen und n=n mod 8.

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:L		B	B	B	B	B	B	B			

BRA Label — Verzweige zu Label (PC-relativ)
.B W — PC+d —> PC
.S

BSET Dn,ea / BSET #K,ea — Bit n Testen und Setzen
B L — Bit-Test —> Z-Flag
Bit = 1

Wenn Source Dn: n=0..31, sonst 0..7. Wenn Destination im RAM, wird immer 1 Byte gelesen und n=n mod 8.

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:L		B	B	B	B	B	B	B			

BSR Label — Call Sub bei Label (PC-relativ)
.B W — PC —> -(SP); PC+d —> PC
.S

BTST Dn,ea / BSET #K,ea — Bit n Testen
B L — Bit-Test —> Z-Flag

Wenn Source Dn: n=0..31, sonst 0..7. Wenn Destination im RAM, wird immer 1 Byte gelesen und n=n mod 8.

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:L		B	B	B	B	B	B	B	B	B	

CHK ea,Dn — Register gegen Limits checken
if Dn <0 or Dn >(ea) then trap

W

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:W		W	W	W	W	W	W	W	W	W	W

CLR ea
B W L

Lösche Operand
0 —> D

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:*		*	*	*	*	*	*	*			

CMP ea,Dn
B W L

Vergleiche Operanden
Flags wie nach D minus S

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:*	WL	*	*	*	*	*	*	*	*	*	*

CMPA ea,An
W L

Vergleiche Adressen
Flags wie nach D minus S

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:*	*	*	*	*	*	*	*	*	*	*	*

Wort-Operand wird vorher auf Long erweitert

CMPI #K,ea
B W L

Vergleiche gegen Konstante
Flags wie nach D minus S

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:*		*	*	*	*	*	*	*			

CMPM (An)+,(An)+
B W L

Vergleiche Speicherstellen
Flags wie nach D minus S

DBcc Dn,Label

Teste cc. Dekrementiere Dn. Branch
if cc = false then Dn=Dn-1
if Dn <> -1 then BRA Label
else »hier weiter«

DIVS ea,Dn
W

Dividiere Worte Signed
D/S —> D

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:*		*	*	*	*	*	*	*	*	*	*

Quotient im niederwertigen Wort, Rest im höherwertigen

DIVU ea,Dn — Dividiere Worte Unsigned
W — D/S —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*	*	*	*

Quotient im niederwertigen Wort, Rest im höherwertigen

EOR Dn,ea — Logisch XOR
B W L — S xor D —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

EORI #K,ea — Logisch XOR mit Konstante
B L — S xor D —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

EORI #K,CCR — XOR Konstante mit CCR
B — S xor CCR —> CCR

EORI #K,SR — XOR Konstante mit SR !Privileg. !
W — S xor CCR —> CCR

EXG Rn,Rn — Tausche Register
L — Rn <—> Rn

EXT Dn — Dn vorzeichenrichtig erweitern
W L

ILLEGAL — löst Illegal-Exception aus

JMP ea — absoluter Sprung (lang)
D —> PC

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*			*	*	*	*	*	*	

JSR ea — absoluter UP-Aufruf

PC —> -(SP); D —> PC

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*			*	*	*	*	*	*	

LEA ea,An — Lade effektive Adresse

L — D —> An

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*			*	*	*	*	*	*	

LINK An,#d — Lokalen Stack einrichten

An —> -(SP); SP —> An; SP+d —>SP

LINK und UNLK werden gebraucht, um eine »linked list« von lokalen Variablen für verschachtelte UP-Aufrufe anzulegen

---

LSL Dn,Dn / LSL #K,Dn / LSL ea — Logisch links schieben

B W L — D n Bits geschoben —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			W	W	W	W	W	W	W			

0 wird nachgeschoben, herausgeschobenes Bit geht in das C- und X-Flag

---

LSR Dn,Dn / LSR #K,Dn / LSR ea — Logisch rechts schieben

B W L — D n Bits geschoben —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			W	W	W	W	W	W	W			

0 wird nachgeschoben, herausgeschobenes Bit geht in das C- und X-Flag

---

MOVE ea,ea — Kopiere Daten

B W L — D —> S

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:*	WL	*	*	*	*	*	*	*	*	*	*
D:*		*	*	*	*	*	*	*			

MOVE ea,CCR — CCR laden
W — ea —> CCR

Zwar ist das CCR nur 8 Bit breit, bei Bewegung eines Wortes in das CCR wird die obere Worthälfte jedoch ignoriert.

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:*		*	*	*	*	*	*	*	*	*	*

MOVE ea,SR — SR laden ! Privilegiert !
W — ea —> SR

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:*		*	*	*	*	*	*	*	*	*	*

MOVE SR,ea — SR holen ! Privilegiert !
W — SR —> ea

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:*		*	*	*	*	*	*	*			

MOVE USP,An MOVE An,USP — USP holen und laden ! Privilegiert !
L — USP —> An

MOVEA ea,An — Kopiere Adresse
WL — ea —> An

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:*	*	*	*	*	*	*	*	*	*	*	*

MOVEM R_Liste,ea / MOVEM ea,R_Liste — Register-Liste kopieren
W L

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
1:			*		*		*	*	*			
2:			*	*		*	*	*	*	*	*	

1= Register —> Speicher z. B.: movem d1-d3/a1-a4,-(a7)
2= Speicher —> Register z. B.: movem (a7)+,d1-d3/a1-a4

MOVEP Dn,d(An) / MOVEP d(An),Dn — Daten von / zu Peripherie
W L
Daten werden byteweise übertragen

MOVEQ #K,Dn — Übertrage »Quick«
L — #K(8 Bit) —> Dn

MULS ea,Dn — Multipliziere Signed
W — S*D —> D

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:*		*	*	*	*	*	*	*	*	*	*

MULU ea,Dn — Multipliziere Unsigned
W — S*D —> D

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:*		*	*	*	*	*	*	*	*	*	*

NBCD ea — Negiere BCD-Zahl
B — 0-D-X —> D

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:*		*	*	*	*	*	*	*			

NEG ea — Negiere Operand
B W L — 0-D —> D

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:*		*	*	*	*	*	*	*			

NEGX ea — Negiere Operand mit X-Flag
B W L — 0-D-X —> D

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:*		*	*	*	*	*	*	*			

NOP — No Operation
tue nichts (dauert 4 Clock-Zyklen)

NOT ea — Logisch Nicht
B W L — -D —> D (Einer-Komplement)

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

OR ea,Dn / OR Dn,ea — Logisch ODER
B W L — S or D —> D

	DN	AN	(AN)	(AN)+	-(AN)	D(AN)	D(AN,RN)	$.W	$.L	D(PC)	D(PC,RN)	#
S:	*		*	*	*	*	*	*	*	*	*	*
D:	*		*	*	*	*	*	*	*			

ORI #K,ea — Logisch ODER mit Konstante
B W L — #K or D —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

ORI #K,CCR — Odere zu CCR
B — #K or CCR —> CCR

ORI #K,SR — Odere zu SR ! Privilegiert !
W — #K or SR —> SR

PEA ea — Push effektive Adresse
L — D —> -(SP)

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*			*	*	*	*	*	*	

RESET — Rücksetzen ! Privilegiert !
Reset-Leitung für 124 Clock-Zyklen auf 0

ROL Dn,Dn / ROL #K,Dn / ROL ea — Rotiere links
B W L — D n Bits rotiert —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			W	W	W	W	W	W	W			

MS-Bit geht ins LS-Bit und ins Carry-Flag und schiebt links

ROR Dn,Dn / ROR #K,Dn / ROR ea — Rotiere rechts
B W L — D n Bits rotiert —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			W	W	W	W	W	W	W			

LS-Bit geht ins MS-Bit und ins Carry-Flag und schiebt rechts

ROXL Dn,Dn / ROXL #K,Dn / ROXL ea — Rotiere links mit X-Flag
B W L — D n Bits rotiert —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			W	W	W	W	W	W	W			

X geht ins LS-Bit und schiebt links. MS-Bit geht in X und Carry

---

ROXR Dn,Dn / RORL #K,Dn / RORL ea — Rotiere rechts mit X-Flag
B W L — D n Bits rotiert —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			W	W	W	W	W	W	W			

X geht ins MS-Bit und schiebt rechts. LS-Bit geht in X und Carry

---

RTE — Return von Exception — ! Privilegiert !
(Sp)+ —> SR; (SP)+ —> PC

RTR — Return mit Flag
(Sp)+ —> CCR; (SP)+ —> PC

RTS — Return
(SP)+ —> PC

SBCD Dn,Dn / ABCD -(An),-(An) — Subtrahiere BCD
B — D-S-X —> D

Scc — Setze Byte wenn cc true
B

if cc=true $FF —> Byte
else $00 —> Byte

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

STOP #K — Lade SR und Halt — ! Privilegiert !

#K —> SR; Halt bis Interrupt

SUB ea,Dn / ADD Dn,ea — Subtrahiere
B W L

D-S —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	*	WL	*	*	*	*	*	*	*	*	*
D:	*		*	*	*	*	*	*	*			

SUBA ea,An — Subtrahiere Adresse
W L

D-S —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	*	*	*	*	*	*	*	*	*	*	*

Wortoperand wird wie bei EXT.L erweitert

SUBI #K,ea — Subtrahiere Konstante
B W L

D-#K —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

SUBQ #K,ea — Subtrahiere Konstante Quick (#K <=8)
B W L

D-#K —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*	WL	*	*	*	*	*	*	*			

SUBX Dn,Dn / ADDX -(An),-(An) — Subtrahiere mit X-Flag
B W L

D-S-X —> D

SWAP Dn
W

Tausche Worte in Dn
Bit 31..16 <—> Bit 15..0

TAS ea
B

Teste und setze Bit 7 im Byte
Bit 7 —> N/Z-Flag
1 —> Bit 7

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:*		*	*	*	*	*	*	*			

TRAP #n

Trap-Exception
PC —> -(SSP)
SR —> -(SSP)
Vektor n —> PC

TRAPV

Trap, wenn Overflow

TST ea
B W L

Teste Operand gegen Null
Ergebnis in N/Z-Flag

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:*		*	*	*	*	*	*	*			

UNLK An

Unlink
An —> SP; (SP)+ —> An

Die Bedeutung der Condition Codes

	Kürzel	Bedeutung	Deutsch
	CC	Carry Clear	Carry = 0
	CS	Carry Set	Carry = 1
	EQ	Equal	Z =1
	GE	Greater or Equal	>=
***	GT	Greater Than	>
	HI	Higher	>
***	LE	Less or Equal	<
	LS	Less or Same	<=
***	LT	Less Than	<
	MI	Minus	-
	NE	Not Equal	<>
	PL	Plus	+
***	VC	oVerflow Clear	V = 0
***	VS	oVerflow Set	V =1
	T	True	1
	F	False	0

*** Für vorzeichenbehaftete Zahlen

D. Erweiterte ASCII-Tabelle

	32	40C	!	33	41C	"	34	42C	#	35	43C
$	36	44C	%	37	45C	&	38	46C	'	39	47C
(	40	50C	)	41	51C	*	42	52C	+	43	53C
,	44	54C	-	45	55C	.	46	56C	/	47	57C
0	48	60C	1	49	61C	2	50	62C	3	51	63C
4	52	64C	5	53	65C	6	54	66C	7	55	67C
8	56	70C	9	57	71C	:	58	72C	;	59	73C
<	60	74C	=	61	75C	>	62	76C	?	63	77C
@	64	100C	A	65	101C	B	66	102C	C	67	103C
D	68	104C	E	69	105C	F	70	106C	G	71	107C
H	72	110C	I	73	111C	J	74	112C	K	75	113C
L	76	114C	M	77	115C	N	78	116C	O	79	117C
P	80	120C	Q	81	121C	R	82	122C	S	83	123C
T	84	124C	U	85	125C	V	86	126C	W	87	127C
X	88	130C	Y	89	131C	Z	90	132C	[	91	133C
\	92	134C	]	93	135C	^	94	136C	_	95	137C
`	96	140C	a	97	141C	b	98	142C	c	99	143C
d	100	144C	e	101	145C	f	102	146C	g	103	147C
h	104	150C	i	105	151C	j	106	152C	k	107	153C
l	108	154C	m	109	155C	n	110	156C	o	111	157C
p	112	160C	q	113	161C	r	114	162C	s	115	163C
t	116	164C	u	117	165C	v	118	166C	w	119	167C
x	120	170C	y	121	171C	z	122	172C	{	123	173C
|	124	174C	}	125	175C	~	126	176C	Δ	127	177C
Ç	128	200C	ü	129	201C	é	130	202C	â	131	203C
ä	132	204C	à	133	205C	å	134	206C	ç	135	207C
ê	136	210C	ë	137	211C	è	138	212C	ï	139	213C
î	140	214C	ì	141	215C	Ä	142	216C	Å	143	217C
É	144	220C	æ	145	221C	Æ	146	222C	ô	147	223C
ö	148	224C	ò	149	225C	û	150	226C	ù	151	227C
ÿ	152	230C	Ö	153	231C	Ü	154	232C	¢	155	233C
£	156	234C	¥	157	235C	ß	158	236C	ƒ	159	237C
á	160	240C	í	161	241C	ó	162	242C	ú	163	243C
ñ	164	244C	Ñ	165	245C	ª	166	246C	º	167	247C
¿	168	250C	⌐	169	251C	¬	170	252C	½	171	253C
¼	172	254C	¡	173	255C	«	174	256C	»	175	257C
ã	176	260C	õ	177	261C	Ø	178	262C	ø	179	263C
œ	180	264C	Œ	181	265C	À	182	266C	Ã	183	267C
Õ	184	270C	¨	185	271C	´	186	272C	†	187	273C
¶	188	274C	©	189	275C	®	190	276C	™	191	277C
ĳ	192	300C	Ĳ	193	301C	א	194	302C	ב	195	303C
ג	196	304C	ד	197	305C	ה	198	306C	ו	199	307C
ז	200	310C	ח	201	311C	ט	202	312C	י	203	313C
כ	204	314C	ל	205	315C	מ	206	316C	נ	207	317C
ס	208	320C	ע	209	321C	פ	210	322C	צ	211	323C
ק	212	324C	ר	213	325C	ש	214	326C	ת	215	327C
ן	216	330C	ך	217	331C	ם	218	332C	ף	219	333C
ץ	220	334C	§	221	335C	∧	222	336C	∞	223	337C
α	224	340C	β	225	341C	Γ	226	342C	π	227	343C
Σ	228	344C	σ	229	345C	µ	230	346C	τ	231	347C
Φ	232	350C	Θ	233	351C	Ω	234	352C	δ	235	353C
∮	236	354C	ϕ	237	355C	∈	238	356C	∩	239	357C
≡	240	360C	±	241	361C	≥	242	362C	≤	243	363C
⌠	244	364C	⌡	245	365C	÷	246	366C	≈	247	367C
°	248	370C	•	249	371C	·	250	372C	√	251	373C
ⁿ	252	374C	²	253	375C	³	254	376C	¯	255	377C

Stichwortverzeichnis

U

V

W

X

Z

Atari ST Modula-2 Programmierhandbuch

Die Autoren:

STEFAN DÜRHOLT studiert Informatik, Mathematik und Physik an der Universität Karlsruhe. Als freier Mitarbeiter für ein Wuppertaler Softwareunternehmen hat er an der Entwicklung komplexer Softwarepakete mitgearbeitet.

JOCHEM SCHNUR ist als Studiendirektor für Mathematik, Physik und Informatik an einem Wuppertaler Gymnasium tätig. Er verfaßte bereits ein Lehrbuch über Pascal.

Modula-2 ist eine Weiterentwicklung der Sprache Pascal. Ihr herausragendes Merkmal ist die Unterstützung der schrittweisen strukturierten Programmierung in separaten Einheiten, den Modulen, die getrennt übersetzt, geprüft und anschließend zu einem Programm zusammengefügt werden. Der Anwender kann auf diese Weise leicht eigene Programmbibliotheken erstellen. Darüber hinaus enthält die Sprache maschinen- und systemnahe Elemente, was insbesondere dem Atari-Programmierer elegante Möglichkeiten zum Gebrauch der Betriebssystem- und GEM-Routinen bietet.

Das vorliegende Buch berücksichtigt sowohl den Leser, der sich in diese Sprache einarbeiten will und nur wenig Programmierkenntnisse besitzt, geht aber in starkem Maße auch auf den erfahrenen Programmierer ein, der mit den Sprachen Basic, C oder Pascal bereits vertraut ist. Beginnend mit grundlegenden Programmiertechniken wird der Leser schnell an professionelle Module für den betreffenden Themenkreis herangeführt. Die einzelnen Sprachelemente werden dabei anhand vieler, interessanter Beispiele dargestellt. So findet man unter anderem Module zur Mandelbrotmenge (Apfelmännchen), zu Simulationen aus Biologie und Physik und ein komplettes Funktionenplotprogramm, das Konzepte der künstlichen Intelligenz aufzeigt. Daneben werden in vielen Anwendungssituationen Atari-interne Sachverhalte erklärt. Die Beschreibungen sind durch zahlreiche Abbildungen illustriert und geben viele Anregungen zur eigenen Programmierung.

Aus dem Inhalt:
- Spracheinführung
- Behandlung wichtiger Datenstrukturen
- Benutzung des 68000-Assemblers unter Modula-2
- GEM-Programmierung unter Modula-2
- Entwicklung eines komplexen Programmpaketes unter Modula-2

und vieles mehr.

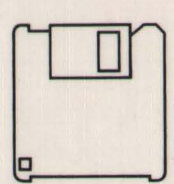
Die Begleitdisketten:
- Die zwei doppelseitigen Disketten enthalten in 150 Modulen alle Programmbeispiele, teilweise auch in kompilierter Form.

Hardware-Anforderungen:
- Atari der ST-Serie
- Schwarzweiß- oder Farbmonitor, doppelseitiges Laufwerk

Software-Anforderungen:
- Modula-2 Compiler, zum Beispiel Megamax Modula-2

ISBN 3-89090-775-X

Markt&Technik





DM 69,-
sFr 63,50
öS 538,-